

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

**Інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу**

До захисту допущено:

Завідувач кафедри

_____ Оксана ТИМОЩУК

«___» _____ 20__ р.

**Дипломна робота
на здобуття ступеня бакалавра
за освітньо-професійною програмою «Системний аналіз і управління»
спеціальності 124 «Системний аналіз»
на тему: «Програмна система для підтримки розподілених даних по
управлінню завантаженістю готельного комплексу»**

Виконав:

студент IV курсу, групи КА-74

Поліщук Артем Богданович _____

Керівник:

доцент кафедри ММСА,

канд. фіз.- мат. наук Яковлева А.П. _____

Консультант з економічного розділу:

доцент, к.е.н.

Рощина Н.В.

Консультант з нормоконтролю:

доцент, к.т.н., доцент кафедри ММСА

Коваленко Анатолій Єпіфанович _____

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

Студент _____

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 124 «Системний аналіз»

Освітньо-професійна програма «Системний аналіз і управління»

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

_____ Оксана ТИМОЩУК

«25» травня 2021 р.

ЗАВДАННЯ

на дипломну роботу студента

Поліщука Артема Богданович

1. Тема роботи «Програмна система для підтримки розподілених даних по управлінню завантаженістю готельного комплексу», керівник роботи канд. фіз.-мат. наук Яковлева А.П., затверджені наказом по університету від «__» _____ 2021 р. № _____
2. Термін подання студентом роботи 07.06.2021 р.
3. Вихідні дані до роботи: предметна область у вигляді основних сутностей готельного комплексу та вимог до його адміністрування
4. Зміст роботи: розробка програмної системи для підтримки розподілених даних по управлінню завантаженістю готельного комплексу
5. Перелік ілюстративного матеріалу: презентація
6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Рощіна Н.В., доцент кафедри ТПЕ		

7. Дата видачі завдання: 21.02.2021

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Затвердження теми ДР	14.02.2021 – 21.02.2021	виконано
2.	Ознайомлення зі структурою БДР відповідно до Положення про державну атестацію студентів НТУУ «КПІ ім. І. Сікорського»	21.02.2021 – 14.03.2021	виконано
3.	Аналіз існуючих рішень, що стосуються тематики ДР	14.03.2021 – 05.04.2021	виконано
4.	Проведення аналізу предметної області відповідно до теми БДР та узгодження з керівником	05.04.2021 – 22.04.2021	виконано
5.	Розробка програмного продукту	22.04.2021 – 22.05.2021	виконано
6.	Аналіз отриманих результатів	22.05.2021 – 27.05.2021	виконано
7.	Оформлення пояснювальної записки	22.05.2021 – 27.05.2021	виконано

Студент

Артем ПОЛІЩУК

Керівник

Алла ЯКОВЛЕВА

РЕФЕРАТ

Дипломна робота 116 с., 18 рис., 7 табл., 2 дод., 25 джерела

ПРОГРАМНИЙ КОМПЛЕКС, ПІДТРИМКА ДАНИХ, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, РЕЛЯЦІЙНІ БАЗИ ДАНИХ, ЗАПИТИ SQL, ПРОЕКТУВАННЯ БАЗИ ДАНИХ, ВІДНОШЕННЯ, СИСТЕМИ УПРАВЛІННЯ БАЗАМИ ДАНИХ, СУБД.

Метою роботи є, базуючись на розглянутих методах і підходах, розробити програмний комплекс з підтримки розподілених даних, а саме: розробити архітектуру, спроектувати базу даних, серверну частину та клієнтську частину.

Об'єктом дослідження є програмна система для підтримки розподілених даних по управлінню завантаженістю готельного комплексу.

Предметом дослідження є підходи до повної розробки програмного комплексу.

Актуальність теми полягає в тому, що в умовах сучасного світу бізнесу необхідно мати автоматизовані системи управління і адміністрування, системи такого роду вирішують проблему автоматизації бізнесу.

У результаті виконання роботи було розглянуто основні методи і підходи до проектування кожної частини програмного продукту, проаналізовано найбільш вживані концепції та механізми розробки подібних комплексів. Також було створено веб-додаток, який демонструє один із варіантів реалізації систем з управління даними.

В якості подальших покращень можна реалізувати додаткові функції, такі як: диференціювання відвідувачів за віком, проведення оплати окремо від бронювання, можливість завантажити звітність у файл.

ABSTRACT

Bachelor thesis: 116 p., 18 fig., 7 tabl., 2 append., 25 sources.

SOFTWARE COMPLEX, DATA SUPPORT, CLIENT SERVER ARCHITECTURE, RELATIVE DATABASES, SQL REQUESTS, DATABASE DESIGN, DATABASE RELATIONS, DATABASE MANAGEMENT SYSTEMS, DBMS.

The purpose of the work is, based on the considered methods and approaches, to develop a software package to support distributed data: to develop an architecture, design a database, server part and client part.

The object of the work is a software system to support distributed data on the management of the hotel complex.

The relevance of the topic is that in today's world of business it is necessary to have automated management and administration systems; systems of this kind solve the problem of business automation.

As a result of the work, the main methods and approaches to the design of each part of the software product were considered, the most commonly used concepts and mechanisms for the development of such complexes were analyzed. A web application, which demonstrates one of the options for implementing data management systems, was also created.

As further improvements, additional features can be implemented, such as: differentiation of visitors by age, payment separately from the reservation, the ability to download reports to a file.

ЗМІСТ

ВСТУП	9
РОЗДІЛ 1. Основні концепції і підходи до проектування систем для підтримки розподілених даних	10
1.1. Актуальність розв'язання задачі підтримки даних з управління складною системою	11
1.2. Моделі зберігання даних та доступу до них	11
1.2.1. Документно-орієнтовна модель	12
1.2.2. Реляційна модель	12
1.3. Нормалізація даних у реляційних БД	13
1.4. Основні підходи для забезпечення безпеки і цілісності виконання транзакцій	16
1.5. Об'єктно-реляційний підхід в проектуванні програмного комплексу	20
1.5.1 Архітектура програмних комплексів	20
1.5.2. Об'єктно-орієнтовність як принцип реалізації серверної частини програмного комплексу	22
1.5.3. Об'єктно-реляційне відображення	25
1.6. Постановка задачі, що має бути вирішена в дипломній роботі	28
Висновки до розділу	29
РОЗДІЛ 2. Методи і концепції організації реляційної бази даних з предметної області	31
2.1. Загальні поняття і методи проектування бази даних	31

2.2. Моделювання даних	33
2.2.1. Основні поняття моделювання даних	33
2.2.2. EER діаграма як описання предметної області у реляційній базі даних	35
2.3. Описання таблиць та зв'язків між ними за допомогою механізму ключів	37
2.4. Накладання обмежень з предметної області на реляційну модель за допомогою механізму тригерів	43
Висновки до розділу	47
РОЗДІЛ 3. РОЗРОБКА СИСТЕМИ ПІДТРИМКИ ДАНИХ З УПРАВЛІННЯ ГОТЕЛЬНИМ КОМПЛЕКСОМ	48
3.1. Формування схеми бази даних з предметної області	48
3.2. Накладання додаткових обмежень на схему бази даних	51
3.3. Реалізація запитів до бази даних	54
3.4. Розробка серверної частини програмної системи з підтримки даних для управління готельним комплексом	56
3.4.1. Обрання підходу до програмування	56
3.4.2. Засоби для розробки серверної частини програми	57
3.4.3. Архітектура та сутності програмного комплексу	58
3.4.4. Клієнтська частина та інтерфейс програми	61
Висновки до розділу	64
РОЗДІЛ 4. ФУНКЦІОНАЛЬНО-ВАРТІСТНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ	66

4.1. Постановка завдання	66
4.2. Обґрунтування функцій та параметрів дослідження	66
4.3. Обґрунтування системи параметрів	68
4.4. Експертне оцінювання важливості параметрів	71
4.5. Аналіз варіантів реалізації функцій програмного комплексу	73
4.6. Економічний аналіз варіантів реалізації програмного комплексу	74
4.7. Вибір кращого варіанта ПП техніко-економічного рівня	77
Висновки до розділу	78
ВИСНОВКИ ЗА РОБОТОЮ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ	80
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ	81
ДОДАТОК А (Ілюстративний матеріал доповіді)	84
ДОДАТОК Б (Код програмного продукту)	93
ДОДАТОК В (Список задач, що вирішує програмний продукт)	114

ВСТУП

Важливість систем підтримки даних для успішності бізнесу неможливо переоцінити. Саме тому, проектування таких систем є актуальною проблемою, яка завжди буде потребувати вирішення та знаходження найбільш оптимальних практик.

Об'єкт дослідження: складні системи для підтримки розподілених даних

Предмет дослідження: підходи та концепції проектування систем підтримки розподілених даних

Мета дослідження: створення якісної системи, що буде відповідати вимогам бізнесу

В першому розділі наведено огляд існуючих підходів і рішень в проектуванні систем зазначеного типу.

В другому розділі було розглянуто методи і концепції перенесення предметної області у відношення бази даних.

В третьому розділі було описано процес проектування системи підтримки даних для готельного комплексу.

РОЗДІЛ 1: ОСНОВНІ КОНЦЕПЦІЇ І ПІДХОДИ ПРОЕКТУВАННЯ СИСТЕМ ДЛЯ ПІДТРИМКИ РОЗПОДІЛЕНИХ ДАНИХ

1.1 Актуальність розв'язання задачі підтримки даних з управління складною системою

Проблема управління даними є дуже актуальною ще з давніх часів. Оскільки ведення обліку є невід'ємною складовою в успішному управлінні будь-якою системою, починаючи від готельного комплексу і закінчуючи державами та арміями. Безумовно, настання ери цифрових технологій призвело до значного спрощення підтримки даних. Раніше саме зберігання даних, навіть відносно невеликого об'єму, було нетривіальною задачею, яка потребувала багато паперу. Що стосується доступу до даних, то цифрові технології також сильно спростили виконання цієї задачі. Тепер людство може бути вільним від громіздких архівів з папером, який розсипається від старості, та вільним від необхідності бути фізично присутнім в місці, де дані зберігаються. Але в своїй сутності, проблема зберігання, обробки, управління даними залишилася тою самою, що і до набуття розповсюдження цифрових технологій. Так само, як і раніше, перед нами постає питання: як зберігати дані? Як мати швидкий доступ до них? Як структурувати дані? Більш того, в умовах сучасного бізнесу, архітектори систем для підтримки даних, стикаються з необхідністю гарантувати «гнучкість» системи. Тобто забезпечити можливість відносно простого і дешевого додавання нового функціоналу в систему, у відповідності до вимог замовника. Також, максимально важливим є те, щоб кінцевий користувач мав можливість легко зрозуміти систему та працювати з нею без попередньої підготовки. Таким чином, проблематика проектування системи для підтримки даних полягає у вирішенні наступних задач:

- обрання оптимальної моделі зберігання даних та доступу до них;

- оптимальне структурування даних;
- забезпечення безпеки роботи з даними;
- можливість у майбутньому додавати новий функціонал у систему;
- забезпечення простоти у використанні та інтуїтивного розуміння кінцевим користувачем, який може не мати необхідних навичок та пізнання в сфері комп'ютерних технологій. Таким користувачем, наприклад, може бути працівник готельного комплексу або бухгалтер.

1.2 Моделі зберігання даних та доступу до них

На сьогоднішній день існує багато підходів і моделей зберігання даних та доступу до них. Серед них є:

- ієрархічна модель;
- схема зірки і модель сніжинки;
- модель «сутність-зв'язок»;
- реляційна модель;
- документо-орієнтована модель;
- об'єктно орієнтована модель.

Кожна з цих моделей є достатньо цікавою. Але, якщо говорити саме про проектування бази даних, то найбільшого поширення набула реляційна модель. Ця модель використовує SQL (англ. structured query language), відомий як мова реляційних СУБД [1], для доступу до даних. Вона використовується в абсолютній більшості баз даних. Але, за останній час, широкого розголосу набув NoSQL підхід в проектуванні баз даних. І хоча в практичній реалізації моєї роботи було використано реляційну модель, я вважаю за необхідне розглянути також представника NoSQL підходу, оскільки він є досить цікавим та має великий потенціал. Цей підхід використовується в декількох моделях даних, однією з яких є документо-орієнтована модель. Саме вона буде стисло розглянута як представник NoSQL підходу.

1.2.1 Документо-орієнтована модель

Як очевидно з назви, в основі моделі лежить зберігання даних у вигляді так званих «документів». По-суті, документи є ієрархічними структурами даних, які представлені в вигляді дерев. Дерево містить кореневий вузол, від якого йдуть листові вузли [2]. Оскільки документи представлені у вигляді дерев, то є потенціал для економії пам'яті при зберіганні даних. Бо для того, щоб отримати доступ до деякого вузла, не обов'язково зберігати в оперативній пам'яті всі інші вузли. Це значно збільшує швидкість. Документи можуть формувати колекції, які, у свою чергу, можуть мати інші колекції. Кожен документ має свій індекс – це унікальне значення, яке його характеризує. Найчастіше це уніфіковане ім'я ресурсу (URI), який використовується для запиту. Варто зазначити, що програмний інтерфейс для виконання запитів, а також мова, на якій вони виконуються, можуть дуже сильно відрізнятися, в залежності від СУБД, яка використовується. Як на мене, це є достатньо великим мінусом даного підходу. Оскільки основний його конкурент – реляційна модель, все ж є значно більш уніфікованою. Даний підхід моделі даних є досить потужним та використовується, наприклад, такою корпорацією, як Amazon, що робить його автоматично актуальним.

1.2.2 Реляційна модель

Зафіксувавши існування NoSQL підходу та зробивши короткий огляд одного з його представників, в подальшому ми будемо концентруватися на реляційному підході та використанні SQL. Та будемо розглядати саме цей підхід в якості основного та базового для цієї роботи. Оскільки він є загальноприйнятим та давно перевіреним. Отже, реляційна модель даних – це модель даних, яка

базується не теорії множин. Відповідно до К.Дейта єдина структура даних, яка використовується в реляційних базах, є нормлізоване n -арне відношення, а маніпуляція даними відбувається за допомогою реляційної алгебри, та реляційному обчисленні [3]. Фактично можна сказати, що в реляційних базах даних дані представлені у вигляді таблиць, які мають зв'язки між собою. Проте, тут варто зробити зауваження, що насправді, строго кажучи, таблиця як така, не є відношенням, вона являється всього візуальним представленням відношення.

Як і зазначалося раніше, в якості мови виконання запитів використовується SQL – декларативна мова програмування, яка дозволяє виконувати необхідні операції з таблицями. Важливою перевагою SQL є його незалежність від певної СУБД. На сьогоднішній день, реляційні БД є найбільш поширеним рішенням для зберігання даних в майже будь-якій сфері бізнесу. Не винятками є і системи управління підприємствами. Більшість мов програмування має багато інструментів для інтеграції з реляційними базами даних. Це дозволяє розробляти гнучку архітектуру програмного забезпечення. Таким чином, великою перевагою реляційного підходу можна назвати його популярність і велику кількість інструментів, які допомагають працювати з такими базами.

1.3 Нормалізація даних у реляційних БД

Структурованість даних є дуже важливою складовою при проектуванні систем підтримки даних. Наводячи визначення реляційної моделі, було сказано саме про нормалізоване відношення. Тому необхідно розглянути явище нормалізації та нормальних форм.

Нормалізація – це процес приведення відношень до деякої нормальної форми. Всього виділяють шість нормальних форм та дві додаткові. Процес нормалізації полягає у тому, щоб змусити відношення відповідати умовам, необхідним для набуття деякої НФ. Фактично, умовою для досягнення наступної

нормальної форми є досягнення попередньої плюс виконання додаткових умов. Таким чином, наприклад, відношення не може перебувати в 3НФ попередньо не будучи в 2НФ. Варто зазначити, що процес нормалізації супроводжується розбиттям таблиці на більшу кількість таблиць. Нормалізація далеко не завжди забезпечує кращу швидкість роботи з базою. Проте, безумовно, до деякої міри нормалізація є необхідною умовою для того, щоб вважати базу якісно спроектованою. Важливою задачею нормалізації є усунення збитковості даних, їх повторення. Фактично, це допомагає не дублювати дані, а отже значно зменшує ймовірність того, що з плином часу в системі будуть з'являтися протиріччя. Також це полегшує обслуговування бази та зміну даних в ній. Але занадто серйозна нормалізація може призвести до того, що з такою базою буде дуже складно працювати. Тому що виконання запиту в таку базу призведе до необхідності об'єднання багатьох таблиць, що в свою чергу вже буде негативно впливати на продуктивність. Отже, існують наступні нормальні форми:

Перша нормальна форма (1НФ). Вона є базовою нормальною формою. Відповідно до твердження Крістофера Дейта, в реляційній моделі, відношення завжди перебуває в 1НФ. Але це не стосується таблиць, оскільки ті являються лише відображенням відношень і тому можуть відображати їх некоректно [4]. Отже, для того, щоб таблиця досягала 1НФ необхідне виконання трьох умов:

- першою умовою 1НФ є атомарність. Тобто кожна властивість має одне значення, що відповідає цій властивості;
- другою умовою 1НФ є наявність первинного ключа для кожної таблиці. Тобто, колонки, або декількох колонок, які унікально ідентифікують таблицю;
- кожна група даних, що пов'язані між собою, має бути виділена в окрему таблицю. [5]

Друга нормальна форма (2НФ). Для того, щоб перебувати в 2НФ таблиця має обов'язково перебувати в 1НФ, а також має виконуватись ряд додаткових умов:

- якщо деякі підмножини даних повторюються в різних рядках однієї таблиці, то вони виносяться в окремі таблиць;
- нові таблиці пов'язуються зі старими за допомогою зовнішніх ключів.

Третя нормальна форма (3НФ). Для перебування в третій нормальній формі таблиця має знаходитися в другій нормальній формі, а також має виконуватись наступна додаткова умова: всі поля, які залежать не від первинного ключа, мають бути винесені в окремі таблиці. Достатньо вичерпне визначення цій нормальній формі дав Біл Кент: «кожен неключовий атрибут, має представляти інформацію про ключ, повний ключ і ні про що окрім ключа» [5].

Нормальна форма Бойса-Кодда. Для розуміння НФБК необхідно розуміти поняття функціональної залежності. За своєю сутністю, функціональна залежність – це зв'язок типу «декілька до одного» між множинами атрибутів в середині змінної відношення [6]. Залежна множина атрибутів, або полів, називається залежною частиною. А та, від якої залежать, називається детермінантом. Таким чином, відповідно до цього визначення, для того, щоб відношення перебувало в нормальній формі Бойса-Кодда необхідно:

- перебування відношення в 3 нормальній формі;
- кожен детермінант має бути потенційним ключем.

Четверта нормальна форма (4НФ). Для перебування в 4НФ відношення має виконувати всі умови для перебування в нормальній формі Бойса-Кодда, а також мають бути відсутні багатозначні залежності [7]. Якщо говорити точніше, то всі такі залежності мають бути функціональними.

Окрім наведених вище нормальних форм є ще: п'ята нормальна форма, нормальна форма домен/ключ та шоста нормальна форма. Проте, на практиці, ці форми нечасто зустрічаються. Зазвичай, більшість реляційних баз даних знаходяться у другій або третій нормальних формах. Також іноді зустрічається НФБК та четверта нормальна форма. Але, в цілому, незважаючи на деякі проблем, можливість нормалізації є серйозною перевагою реляційного підходу.

1.4 Основні підходи до забезпечення безпеки і цілісності виконання транзакцій

Транзакція – це група операцій з базою даних, яка розглядається як єдине ціле. В реляційних базах даних, щоб гарантувати валідність транзакцій, проголошується наступна група принципів, які мають виконуватись. Ці принципи мають назву ACID. Наведемо короткий огляд кожного з них.

Atomicity (атомарність) – це властивість транзакції, відповідно до якої вона розглядається як логічна одиниця, навіть якщо складається з декількох запитів. Таким чином виконання цієї властивості гарантує, що, якщо будь яка операція з групи операцій, які входять до транзакції, буде невдалою, то автоматично будуть невдалими всі інші операції. Транзакція може вважатися виконаною тільки, якщо вдало виконана вся множина операцій, що входять до неї.

Consistency (узгодженість) - це властивість, відповідно до якої проголошується, що транзакція може переводити базу даних з одного коректного стану в інший. Тобто, якщо перед виконанням транзакції система була у валідному стані, то і після транзакції мають виконуватись умови усіх обмежень, таких як зовнішні ключі, тригери та інше. Класичним прикладом, що ілюструє принцип узгодженості транзакції може слугувати процес оплати з картки. Для того, щоб оплатити щось картою необхідно спочатку зняти гроші з рахунку користувача, а потім покласти їх на рахунок магазину. Відповідно, в такому разі, транзакція буде мати дві операції. Під час транзакції, коли виконалась перша, але не виконалась друга операція, система перебуває в неправильному стані, але в результаті виконання транзакції валідність системи відновлюється.

Isolation (ізолюваність) – це властивість транзакції, що виконується у багатопоточному середовищі, виконуватись так, і залишати базу після свого виконання у такому стані, наче транзакція була виконана в однопоточному середовищі. Таким чином, можна сказати, що ця властивість проголошує

прагнення до незалежності транзакцій, що виконуються одночасно. Тобто такі транзакції не мають впливати на результат одна одної.

Durability (довговічність) – це властивість, яка проголошує, що результат успішно виконаної транзакції буде збережений навіть у випадку системних збоїв. Таким чином, якщо деякий користувач отримав інформацію від системи про те, що його транзакція пройшла успішно, а після цього, внаслідок настання деякої проблеми, робота системи припинилася, або була некоректною, то після відновлення коректної роботи системи, результат транзакції, що виконалася до збою, має бути повністю збережений.

Відповідність цим чотирьом принципам гарантує те, що транзакція буде виконуватись безпечно. Але, якщо забезпечення атомарності та узгодженості не є дуже складною задачею і гарантується більшістю сучасних технологій та вдало спроектованою системою, а довговічність досягається завдяки вчасно проведеному резервному копіюванню даних, то з ізолюваністю можуть виникати серйозні проблеми, а підходи до забезпечення ізолюваності транзакцій можуть суттєво відрізнятися в різних системах.

Проблема ізолюваності полягає в тому, що більшість сучасних систем підтримки даних може працювати у багатопотоковому режимі. Тобто, декілька транзакцій можуть виконуватись одночасно. Це допомагає досягти значно кращої продуктивності системи. Проте, в такому разі збільшується ризик виникнення аномалій, через те, що транзакції можуть змінювати дані, з якими працюють інші транзакції. Але принцип повної ізолюваності досить часто означає дуже значне г роботи системи в цілому. Таким чином, при розробці складних систем підтримки даних, дуже важливо досягти компромісу між швидкістю і належним рівнем ізоляції транзакцій.

Але, перед тим, як розглянути, які ж бувають рівні ізолюваності транзакцій, для початку потрібно сказати, які саме проблеми можуть виникнути під час реалізації будь-якого підходу в проектуванні систем підтримки даних, які

не виключають перспективу роботи одночасно у багато потоків. Отже, традиційно виділяється ряд наступних проблем:

- втрачене оновлення (lost update) – коли один і той самий набір даних є одночасно сферою роботи різних транзакцій, то зберігаються тільки останні зміни. Все інше втрачається. Дана проблема є дуже серйозною і її усунення є абсолютно необхідним навіть в системах, де може не досягатися повний рівень ізольованості транзакцій;
- «брудне читання» (dirty read) – це проблема, коли деяка транзакція під час свого виконання може зчитати дані, змінені іншою транзакцією, яка після цього відкатиться. Таким чином, транзакція, яка зчитала дані, буде мати їх невалідну версію. Це призведе до виникнення аномалій в системі. Таким чином, дана проблема є досить критичною і потребує усунення;
- читання, що не повторюється (non-repeatable read) - це проблема, яка виникає, коли деяка транзакція зчитує одні і ті самі дані декілька разів, під час свого виконання. Але в період між зчитуваннями дані виявляться зміненими, таким чином, після кожного зчитування, транзакція працює з наборами даних, що відрізняються між собою;
- фантомне читання (phantom reads) – це проблема яка виникає внаслідок того, що деяка транзакція проводить декілька запитів на отримання даних за деякими умовами під час свого виконання, між цими запитами, інша транзакція вносить нові дані, які будуть відповідати поставленим умовами вибору першої транзакції. Таким чином, в різний час, вибір даних однією транзакцією буде мати інший результат. Варто відзначити, що дана проблема є досить схожою на читання, що не повторюється. Але різницею є те, що тут дані не видаляються або змінюються, а додаються нові, фантомні дані.

Описані вище проблеми мають різний рівень впливу на роботу програми і призводять до різної кількості аномалій. Наприклад, фантомне читання часто не є критичною проблемою і багато систем не забезпечують захист від нього, щоб

досягти кращої швидкості виконання транзакцій, що має суттєвий вплив на користувацький досвід. Проте, абсолютна більшість сучасних систем управління базами даних та мов програмування підтримують роботу з транзакціями на різних рівнях ізолюваності, найвищий із яких дозволяє досягти виключення можливості фантомного читання.

Визначаються наступні рівні ізолюваності транзакцій, які використовуються в основних підходах до проектування складних програмних систем і комплексів:

- читання незафіксованих даних (Read uncommitted) – цей рівень ізолюваності фактично є найслабшим. Серед вищезгаданих проблем даний рівень запобігає тільки проблемі втраченого оновлення;
- читання фіксованих даних (Read committed) – це наступний рівень ізолюваності транзакцій, який окрім проблеми втраченого оновлення також запобігає проблемам «брудного» читання. Цей рівень ізолюваності прийнятий за замовченням в деяких СУБД, наприклад MS SQL Server;
- повторюване читання (Repeatable read) – це рівень, що запобігає, окрім усього описаного вище, також проблемі неповторюваного читання. Цей рівень за замовченням встановлений в СУБД MySQL та у фреймворці Spring Data JPA. В якомусь сенсі його можна вважати базовим рівнем ізолюваності, оскільки він надає можливість досягти деякого компромісу між безпекою та швидкістю запису або зчитування даних. Цей рівень є найбільш популярним і використовується в багатьох сучасних рішеннях і підходах;
- впорядкованість (Serializable) – це найвищий рівень ізолюваності транзакцій, який, фактично, дозволяє розглядати виконання паралельних транзакцій, як таких, що виконуються послідовно. Проте, цей рівень значно заповільнює роботу програми, хоча він явно є необхідним у деяких системах. Наприклад, у банківській сфері дуже важливо повністю гарантувати правильність виконання паралельних транзакцій.

1.5 Об'єктно-реляційний підхід в проектуванні програмного комплексу

З написаного вище стає очевидним, що коли мова йде про проектування бази даних, то реляційний підхід є найбільш поширеним та привабливим, на сьогоднішній день. Але, як відомо, будь який програмний комплекс не може складатися тільки з бази даних. Хоча база є неймовірно важливою складовою, програмний комплекс не може існувати без інших шарів програмного забезпечення. Для розуміння об'єктно-реляційного підходу в проектуванні необхідно, для початку, розглянути архітектуру програмного комплексу.

1.5.1 Архітектура програмних комплексів

Більшість серйозних програм в сфері управління мають так звану клієнт-серверну архітектуру. Така архітектура дозволяє тримати базу даних на одному, або декількох серверах, або на вузлах серверів. В свою чергу, користувачі, кожен зі свого пристрою можуть мати доступ до даних та маніпулювати ними, відповідно до власних потреб, у тій мірі, в якій це дозволено програмно. Комунікація з сервером відбувається через браузер (клієнт) за допомогою запитів на програмний інтерфейс (або API). Таким чином. Зі сторони серверної частини відбувається прийом запитів та відправлення відповідей до клієнта. Серверна частина програми обробляє запити від клієнта. Потім вона відправляє запити в базу даних за допомогою SQL, після чого отриманий результат передається назад, клієнту. Безумовно, програми можуть існувати і без серверу додатка. Є можливість відправляти клієнтські запити відразу в базу, наприклад, за допомогою запитів, вбудованих відразу в клієнтський код. Проте, такий підхід видається надзвичайно неефективним, оскільки породжує ряд серйозних проблем. По-перше, він значно зменшує можливість адаптувати програму під нові технології, оскільки користувацький інтерфейс та база даних стають сильно

пов'язаними. Сервер додатку слугує «прокладкою» між базою та кодом клієнта. Це допомагає виокремити всі нюанси взаємодії в окрему частину (якою і є сервер додатку) і дозволяє досягти можливості змінювати технології в майбутньому. Наприклад, якщо буде вирішено перейти з однієї технології front-end на іншу, то не доведеться переписувати всі запити в базу. Достатньо буде тільки налаштувати новий код на роботу зі старим API.

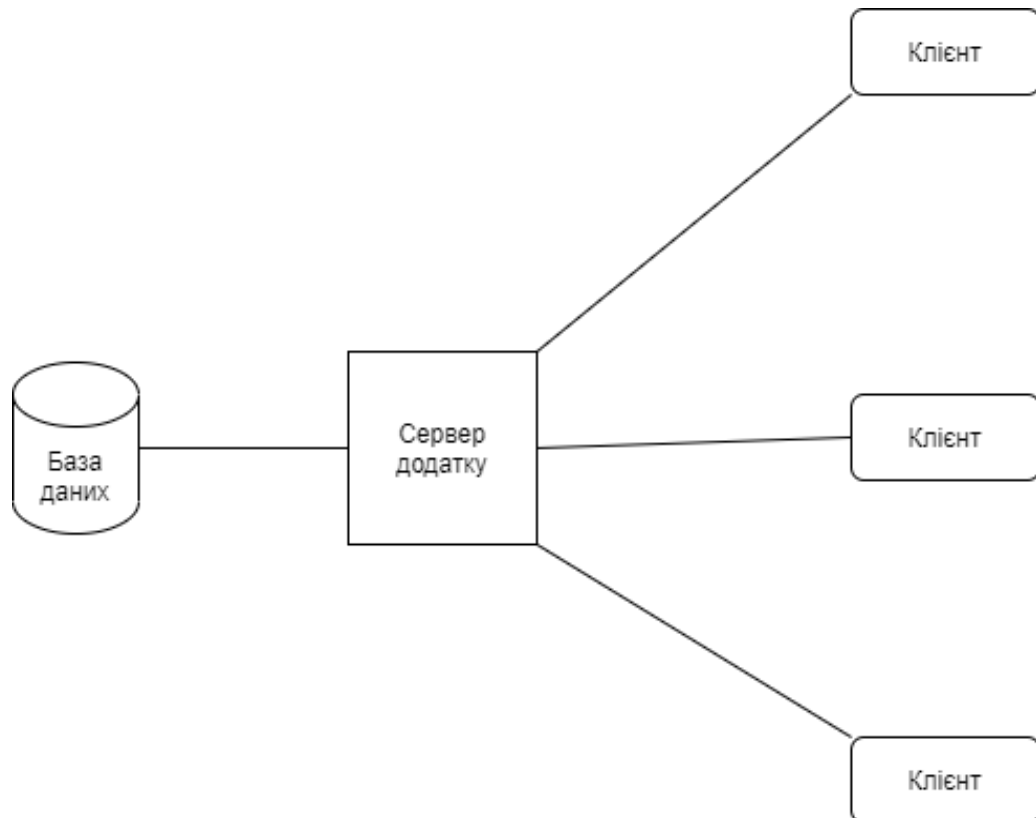


Рисунок 1.1 - Ілюстрація клієнт-серверної моделі

Або, якщо необхідно буде мігрувати базу даних, то не треба буде переписувати клієнтську частину коду. Таким чином, досягається гнучкість.

Також додавання серверу додатку (back-end) спрощує не тільки міграцію на інші технології, а і надає більше можливостей для забезпечення безпеки додатку. Це дозволяє налаштувати перевірку коректності даних не тільки на стороні клієнта, а і на стороні сервера, що значно покращує перспективу безпечного використання додатку.

Введення серверної частини дозволяє архітектурі значно краще відповідати такому базовому принципу програмування, як SRP (single-responsibility principle) – принципу єдиної відповідальності. Тобто, клієнтський код виконує тільки свій прямий обов’язок – взаємодію з клієнтом, тоді як особливості реалізації запитів в базу його не стосуються.

Таким чином, клієнт-серверна архітектура робить програму значно чистішою та кращою. Саме тому, проводячи огляд існуючих рішень, варто зупинити увагу на такому підході, оскільки він є максимально привабливим.

1.5.2 Об’єктно-орієнтовність як принцип реалізації серверної частини програмного комплексу

Маючи обґрунтування необхідності реалізації серверної частини програмного комплексу, необхідно зазначити, що абсолютна більшість таких рішень, реалізована за допомогою об’єктно орієнтовних мов програмування (далі ООП). Серед таких мов основними є Java та C#. Суть ООП полягає в представленні програмних сутностей в вигляді об’єктів. Об’єкти є екземплярами (тобто конкретними реалізаціями) деяких класів. Таким чином, клас – це шаблон, який описує об’єкти, представлені в програмі. Для прикладу, можна сказати, що в програмі готельного комплексу об’єктами будуть Користувач, Оплата, Кімната. ООП має деякі принципи, серед яких є принцип абстракції – виокремлення необхідних саме в цій програмі характеристики об’єкта. Тобто, для прикладу, об’єкт Користувач в програмі готельного комплексу буде мати атрибут «ім’я», але скоріше за все не буде мати атрибуту «група крові». Це дозволяє розробляти програму в категоріях, які інтуїтивно зрозумілі будь-якій людині.

Також, ООП допомагає виконувати ледь не основну задачу програмування, а саме - досягання гнучкості і можливості розширення функціоналу в

майбутньому. Завдяки правильному використанню основних принципів ООП можна досягти такого рівня абстракцій в коді, що подальше розширення функціоналу не буде викликати особливих труднощів.

Варто розглянути основні принципи ООП, так як вони формують основу сучасного підходу до розробки складних програмних комплексів. Крім абстракції такими принципами є:

- а) Інкапсуляція – це принцип відповідно до якого дані, тобто властивості об'єктів, мають бути закритими для доступу з інших об'єктів. Цей принцип реалізовується за допомогою того, що властивості класів робляться закритими, тоді як для роботи з об'єктом надається інтерфейс для взаємодії з ним. Таким чином, змінювати об'єкт, або «контактувати» з ним можливо тільки за заданими в самому об'єкті правилами (методами). Це дозволяє робити об'єкти незалежними один від одного, та не розкривати особливості реалізації іншим об'єктам. Такий підхід покращує гнучкість системи. Тобто, якщо деякий об'єкт має доступ до особливостей реалізації іншого об'єкту, то при зміні особливостей цей об'єкт теж буде потребувати редагування. А при великій системі такі речі дуже складно відслідковувати і це буде призводити до помилок. Якщо ж робота з об'єктом відбувається за допомогою визначених правил, то це дозволяє змінювати конкретні особливості без зміни цих правил. Це є серйозною перевагою ООП та дозволяє досягти кращого рівня абстракцій та незалежності сутностей;
- б) Наслідування (успадкування) – це властивість програмних сутностей включати в себе характеристики інших об'єктів, а також додавати до них нові. Цей принцип дозволяє формувати ієрархії об'єктів, що є дуже важливим для забезпечення доступу до даних та до взаємодії між об'єктами. Використання наслідування дозволяє також значно скоротити кількість коду, що повторюється;
- в) Поліморфізм – це надання однакового інтерфейсу екземплярам різних класів. Найчастіше цей принцип поєднується з успадкуванням, або явищем

реалізації інтерфейсів у мовах, які це підтримують. Цей принцип дозволяє звертатися до об'єкту дочірнього типу, як до об'єкту типу батьківського. Цей принцип є надзвичайно потужним, оскільки він надає можливість програмному коду максимально абстрагуватись від особливостей конкретних реалізацій програмних сутностей. Це дозволяє частинам програми бути незалежними та взаємозамінними. Безумовно, також це допомагає у майбутньому, коли необхідно додавати новий функціонал.

Правильне використання основних принципів об'єктно-орієнтовного програмування робить програмну систему неймовірно потужною в плані гнучкості та чистоти коду. Також позитивною рисою такого підходу можна вважати те, що саме поняття об'єкту є близьким для кожної людини. Це дозволяє оперувати програмними сутностями як об'єктами з реального життя і значно полегшує програмування.

Проте, варто зазначити, що даний підхід має ряд мінусів. Нажаль, правильне розуміння принципів ООП, потребує серйозної експертизи, тоді як необережне використання даної моделі призводить не тільки до великої збитковості даних, а і до значного збільшення складності. Такий код може бути важко зрозуміти, що дуже ускладнює командну розробку. Також, для прозорості треба сказати, що, хоча ООП досі залишається найбільш популярним варіантом, в останні часи він піддається критиці, а основні об'єктно-орієнтовні мови відходять від чистого ООП до таких парадигм, як функціональне програмування. Але, не зважаючи на те, що чистий ООП в наші часи не є обов'язковим. Оскільки принцип успадкування часто поступається імплементації інтерфейсів, а деякі прості програми можна реалізувати взагалі без поліморфізму і наслідування. Та все одно сама концепція представлення програмних сутностей у вигляді об'єктів залишається основною в сфері розробки складних систем. Вона довела свою життєздатність, хоча і вимагає дуже серйозного ставлення та великого рівня експертних знань програміста.

Таким чином, як реляційна модель є загальноприйнятою та найбільш оптимальною для проектування баз даних, так і ООП є найкращим вибором для проектування тої частини програми, яка, власне, буде взаємодіяти з базою, хоча такий підхід і не застрахований від недоліків.

1.5.3 Об'єктно-реляційне відображення

Так як серверна частина (об'єктна) взаємодіє з базою, яка є реляційна, то постає питання: як забезпечити вдалу взаємодію між цими двома підходами. Так, ми знаємо, що існують об'єктно-орієнтовані моделі даних, але вони не набули використання. Тобто необхідно вирішувати проблему того, як же правильно передавати дані від програми до бази та назад. Тут постають достатньо серйозні проблеми, які варто вирішувати. Для достатньо простих систем відображення не є чимось складним. Як правило, сам по собі об'єкт достатньо просто перенести в таблицю. Наприклад, деякий об'єкт «Користувач» з полями «ім'я», «телефон» і «поштовий індекс» легко переноситься в відповідну таблицю з такими самими атрибутами. Але, при більш серйозних системах виникає ряд проблем, вирішення яких є нетривіальною задачею. Ці проблеми формують парадигму «невідповідності», постулатом якої є те, що об'єктна та реляційна моделі не дуже добре взаємодіють. Серед основних проблем об'єктно-реляційної взаємодії є наступні:

- проблема відображення наслідування в реляційних БД. Як відомо, наслідування є однією з базових властивостей об'єктної парадигми. В свою чергу, реляційна модель не має тривіальних механізмів відображення наслідування;
- різні способи доступу до даних для парадигм. В об'єктній парадигмі доступ до інформації про об'єкт відбувається за допомогою послідовного переходу від батьківського об'єкта до дочірнього. В свою чергу даний

спосіб отримання даних з реляційної БД є не просто неоптимальним, а і досить небезпечним. Так як, при не достатньому рівні ізоляції транзакцій і при великій кількості запитів для виконання однієї операції, є дуже велика ймовірність втрати даних або утворення різноманітних аномалій. Тому реляційний підхід постулює необхідність мінімальної кількості запитів для отримання цільової інформації, наприклад за допомогою JOIN;

- парадигма ООП підтримує два види перевірки рівності об'єктів. Це порівняння за посиланням, тобто перевірка того, що об'єкти зберігаються в тому самому місці пам'яті, а тому є рівними. А також логічне порівняння, яке найчастіше являє собою перевірку логічної рівності властивостей об'єкта (тобто його полів), також рівність так званих хеш-кодів – деякого унікального (майже завжди) числа, що рахується на основі властивостей об'єкта. В свою чергу, в реляційному підході є тільки один спосіб перевірки рівності записів – первинний ключ, який унікально визначає кожен запис таблиці;
- реляційна модель підтримує можливість визначення двонаправленого зв'язку між відношеннями. Це виконується за допомогою зовнішніх ключів (foreign key). В свою чергу, в об'єктній парадигмі неможливо визначити такий тип зв'язку. Для відображення двонаправленого зв'язку необхідно створити два однонаправлені посилання в кожному з об'єктів. При цьому, ці зв'язки будуть фактично незалежними, тобто належна взаємодія, яка в реляційній парадигмі досягається дуже просто, в об'єктній потребує значних зусиль і обережності;
- часто представлення об'єкта є значно менш детальним ніж відношення, відображене таблицею в реляційній базі. Тому, для зберігання одного об'єкта в реляційній базі, може вимагатися створення декількох таблиць. Це теж може ускладнити роботу з базою.

Таким чином, розглянувши проблематику об'єктно-реляційного відображення, можна побачити, що кожна з парадигм має свої позитивні

характеристики, але кожна з них також не позбавлена проблем. Ці проблеми особливо загострюються при необхідності трансформувати дані з однієї моделі в іншу.

Власне, задачу відображення даних між парадигмами і має вирішувати типова реалізація ORM (object-relational mapping), тобто об'єктно реляційного відображення. Про що, власне, і говорить сама його назва. Безумовно, саме по собі ORM може набувати абсолютно різного вигляду при різних вихідних умовах. Звичайно, виходячи з описаних вище проблем, як таке ORM завжди буде нести в собі ряд компромісів. Якщо говорити про основні сучасні ORM, то я б назвав відомий Hibernate, який використовується дуже часто, та лежить в основі потужного фреймворку Spring Data JPA (Java Persistence API). Hibernate та подібні до нього технології дозволяють сучасному розробнику не особливо «дивитися» в базу даних. Варто зазначити, що ORM допомагає виконанню принципу SRP, тобто дозволяє кожній частині програми робити тільки ту роботу, яка є безпосередньо її основною ціллю. Таким чином типовий серверний код, який використовує ORM може не містити в собі нативних SQL запитів. Запити можуть писатися об'єктно орієнтовною мовою, а Hibernate, або інше з існуючих рішень, трансформує ці запити в зрозумілі для реляційної моделі SQL запити.

Проте, необхідно зазначити, що використання повної потужності таких технологій, на мою думку, не є обов'язковим каноном ORM, власне як і не завжди є поганим написання безпосередньо нативних SQL запитів прямо з об'єктного коду. Так, для програмних систем з не дуже великим об'ємом, хорошим рішенням може бути написання звичайних запитів, це допомагає оптимізувати роботу програми. Оскільки використання об'єктно-орієнтовних мов для запитів призводить до більш повільного виконання, так як програмі необхідно трансформувати це в SQL. Таким чином, на мою думку, непоганим варіантом може слугувати використання наявних реалізацій ORM для того, щоб позбавитися написання великої кількості чисто технічного коду, який відповідає за встановлення зв'язку з базою та оброблення відповіді з неї. Такий код з

легкістю можна замінити декількома рядками з Spring Data Jpa. Але при цьому не варто повністю відмовлятися від написання SQL запитів, особливо складних.

Отже, ORM є необхідною складовою для взаємодії об'єктної та реляційної парадигм. Сучасні технології дозволяють повністю відмовитися від написання SQL запитів у звичайній формі, що значно полегшує роботу програміста та прискорює розробку застосунку. Проте, потужність технологій по типу Spring Data JPA полягає в тому, що вони надають можливість писати звичайні запити для того, щоб оптимізувати роботу з базою. Таке використання технологій, на мою думку, є золотою серединою, коли мова йде про проектування складних систем середнього об'єму. Це дозволяє позбавитись повторювань технічного коду, який бере на себе фреймворк, але в той же час на дає гнучкості і покращує управління системою. Серед мінусів такого «компромісного» підходу є те, що програміст зобов'язаний мати серйозну експертизу для того, щоб написати достатньо оптимальні запити і не погіршити роботу системи.

1.6 Постановка задачі, що має бути вирішена в дипломній роботі

Задачею роботи є, на основі проведеного аналізу основних концепцій і підходів, розробити архітектуру та реалізувати програмний продукт, який буде вирішувати проблему підтримки даних з управління готельним комплексом. Тобто, фактично, буде являти собою систему адміністрування готелю. Даний продукт має містити:

- реляційну базу даних з відношень, які сформовані відповідно до предметної області;
- серверну частину програми, яка відповідає за передачу даних з клієнта до бази даних, та реалізована за допомогою об'єктно-орієнтовного підходу;
- клієнтську частину, яка реагує на дії кінцевого користувача і направляє запити на сервер.

Висновки до розділу

В даному розділі було коротко розглянуто необхідність у системах підтримки даних для управління складними системами. Було розглянуто ряд вимог до такого роду систем. Можна зробити висновок, що такі системи необхідні для нормального функціонування сучасного бізнесу, та для контролю за ним.

Основним принципом формування таких систем є принцип клієнт-серверної архітектури, при якій кінцевий користувач взаємодіє з сервером за допомогою клієнта (тобто браузера). В свою чергу серверна частина взаємодіє з базою даних.

Проектування бази даних є вагомим фактором в розробці системи підтримки даних. В цьому розділі коротко було розглянуто основного конкурента реляційній моделі, тобто NoSQL підхід в варіанті документо-орієнтованої моделі. Але реляційна модель має ряд дуже серйозних плюсів, тому вона розглядається в якості основної.

Однією з основних характеристик реляційної моделі є явище нормалізації. Було розглянуто 4 основні нормальні форми, а також одна додаткова форма (нормальна форма Бойса-Кодда).

У цьому розділі було розглянуто основні проблеми, які виникають під час роботи програмних комплексів з транзакціями у паралельному режимі. Також було розглянуто рівні ізолюваності транзакцій, які запобігають деяким, або всім з цих проблем.

Було наведено характеристику основного підходу в реалізації серверної частини системи. Таким підходом є об'єктно-орієнтовне програмування, яке полягає в представленні програмних сутностей у вигляді об'єктів, які описуються за певними шаблонами (класами).

Оскільки представлення даних в БД (реляційна модель) та представлення даних в програмі сервера додатку (об'єктна модель) – це різні парадигми, то важливо досягти максимально коректного відображення даних між цими двома моделями їх представлення. Підхід, який використовується для вирішення цієї проблеми має назву ORM (object-relational mapping) – об'єктно-реляційне відображення. Такий підхід збільшує гнучкість та покращує швидкість роботи системи, тобто дозволяє оптимізувати роботу.

РОЗДІЛ 2: МЕТОДИ І КОНЦЕПЦІЇ ОРГАНІЗАЦІЇ РЕЛЯЦІЙНОЇ БАЗИ ДАНИХ З ПРЕДМЕТНОЇ ОБЛАСТІ

2.1 Загальні поняття і методи проектування бази даних

Для початку треба визначити, що ж таке проектування бази даних. Т. Коннолі і К.Бегг у своїй книзі «Бази даних. Проектування, реалізація і підтримка. Теорія і практика» визначають проектування бази даних наступним чином: «Проектування бази даних – це процес створення проекту бази даних, яка призначена для підтримки функціонування підприємства і яка допомагає досягненню його цілей» [8]. Таким чином, основною метою проектування є забезпечення можливості максимально успішного функціонування підприємства. Проектування бази даних – це дуже важливий етап розробки програмного комплексу з підтримки даних. Це важливо, тому що від вдалого проектування залежать такі фактори, як легкість подальшої розробки, можливість розширення майбутньої системи новими функціональними блоками, модулями, підсистемами. Також, вдале проектування бази даних є надзвичайно важливим для подальшої підтримки і усунення помилок, без яких не можна обійтися повністю. Таким чином, успішне проектування бази може закласти підвалини для подальшого успіху розробки, а невдале проектування призведе до значних ускладнень і, як наслідок, втрати коштів. Тому, до проектування бази даних треба ставитися дуже серйозно. Дослідниками виділяється два основні підходи в проектуванні бази даних: висхідний та низхідний [8].

Висхідний підхід полягає у декомпозиції предметної області, починаючи, від найнижчого рівня, тобто фактично починаючи з аналізу сутностей і зв'язків і лише потім переходячи у відношення самої бази даних. При такому підході першочерговим можна вважати процес складення необхідних атрибутів для виконання задачі. Лише потім проводиться нормалізація яка являє собою в деякому роді оптимізацію раніше визначених зв'язків. Цей підхід є досить цікавим і, очевидно, добре підходить для формування бази даних з невеликої предметної області (як це є у даній роботі). Але при великій кількості

потенційних зв'язків і сильно ускладненій системі є ризик виникнення плутанини і ускладнень при проектуванні.

Низхідний підхід краще задовольняє потреби у проектуванні саме складних баз даних з великою кількістю потенційних відношень і зв'язків між ними. Сенс підходу полягає у визначенні найкрупніших сутностей, які «лежать на поверхні» і подальшої декомпозиції з заданням необхідних подробиць.

Існують також і інші підходи до проектування бази даних, які, певною мірою є узагальненнями вищенаведених. Наприклад «від загального до окремого». Він полягає, як і висхідний підхід, у початковому визначенні сутностей і зв'язків між ними, але при цьому, подальше проектування пов'язане з додаванням сутностей, які взаємодіють з уже наявними, в свою чергу, у висхідного підходу така взаємодія не є необхідною. Також цікавим методом проектування бази даних може слугувати «змішана стратегія проектування», який одночасно використовує обидві основні методи, тобто висхідний і низхідний. Даний спосіб проектування полягає у тому, що формуються різні частини моделей за допомогою двох методів. Після чого, ці частини мають бути об'єднані для утворення моделі, яка являє собою результат.

Незважаючи на те, що існують чудові та дуже гарно описані методи проектування баз даних, інформація про які наведена вище, варто зазначити, що всі вони являють собою більше емпіричний підхід. Безумовно, на даному етапі конструювання системи, неможливо обійтися без суб'єктивного підходу. Але, саме тому що чіткі правила коректного проектування визначити достатньо складно або навіть неможливо, дана задача вимагає не тільки серйозних теоретичних знань спеціаліста, а також наявності у нього досвіду у подібній практиці, а це робить дану задачу нетривіальною.

2.2 Моделювання даних

2.2.1 Основні поняття моделювання даних

Після завершення етапу проектування бази даних, тобто після декомпозиції предметної області у відношення реляційної бази даних, наступним етапом є моделювання даних. Моделювання – це в деякому роді складання документації до новоствореної схеми бази даних. Модель має розкривати сутність відношень та зв'язків між ними і слугувати «інструкцією» розробникам, які будуть створювати базу даних, а також тим, хто буде програмувати серверну частину програми. Документація є необхідною для успішної розробки, саме тому постає задача якісного формування моделі даних. Безумовно, в ході виконання розробки бази даних, а потім і серверної частини програмного комплексу з підтримки розподілених даних, і навіть клієнтної частини, будуть виникати питання пов'язані з вимогами до даних. Саме тому дуже необхідно мати чітке формулювання таких вимог. До того ж розбиття предметної області в схемі бази даних потенційно може змінюватись в разі зміни задачі, має бути надійно закріплене в документації. Це важливо, тому що сутності, які формують таблиці, зв'язки між ними і поля таблиць, які потім перейдуть в об'єктну модель також, мають фундаментальне значення для системи. Але, якщо ці аспекти не будуть належним чином відображені в документації, то це призведе до серйозних наслідків і проблем в розробці. Моделювання значно спрощує розуміння наступних нюансів, які виникають в процесі роботи з даними:

- характер даних;
- використання даних відповідно до предметної області, тобто проблема встановлення правильних меж використання;
- вимоги, які поставлені до даних відповідно до предметної області та завдання.

Також традиційно виділяють наступні вимоги для моделі даних [8]:

Таблиця 2.1 - Вимоги до моделі даних

Показник	Характеристика показнику
----------	--------------------------

Структурна достовірність	Відповідність способу визначення і організації інформації на даному підприємстві
Цілісність	Узгодженість з методом використання і управління інформацією в підприємстві
Відсутність збитковості	Будь-яка частина даних має бути представлена тільки один раз, тобто необхідно досягти виключення зайвої інформації
Можливість розширення в майбутньому	Можливість легко адаптувати модель під нові вимоги, які можуть виникнути
Можливість сумісного використання	Відсутність належності до якогось конкретного додатку або технології. Тобто необхідно гарантувати можливість використання моделі в багатьох програмних комплексах, а не тільки в одному
Виразність	Необхідність чіткого представлення даних, зв'язків між ними, відмінностей
Простота	Модель має бути легкодоступною для розуміння, як професіоналами, так і людьми, які не мають достатнього рівня експертизи
Схематичне представлення	Необхідність представлення моделі у вигляді схематичного зображення

Варто зазначити, що дані критерії у відомому сенсі представляють ідеал моделі даних. Навряд можна створити модель, яка буде відповідати кожному з даних критеріїв. Саме тому, як і у випадку з нормалізацією та ізолюваністю транзакцій, вдала модель представляє собою деякий компроміс, ціллю якого є досягання максимальної кількості даних характеристик та вимог. Виходячи з вищенаведеного, на мою думку, чудовим варіантом моделі даних є ER-модель, про яку і буде йти мова в наступному підрозділі.

2.2.2 EER діаграма як описання предметної області у реляційній базі даних

Перед тим як переходити до описання сутності EER (Enhanced entity-relationship) діаграми, дуже важливо розуміти принцип ER-моделі (entity-relationship model). Це так, оскільки сама по собі EER модель повністю заснована на ER-підході і лише є невеликим його доповненням, а різниця між ними не суттєва для цілей даної роботи[8]. Саме тому, зрозумівши базис і надавши йому належний опис, не буде виникати складнощів із розумінням EER-моделі.

ER-моделювання – це підхід, який моделює базу даних за допомогою визначення найважливіших даних, які називаються сутностями (або entity), виокремленням зв'язків між сутностями, які представлені в моделі. Після цього етапу модель доповнюється додатковими даними, наприклад, інформацією про сутності і зв'язках, вказуються обмеження, які мають вплив на сутності та взаємодію між ними. Варто сказати, що підхід ER-моделювання є найбільш популярним, тому його розуміння є надважливим для загального розуміння проблематики проектування баз даних та для успішного виконання даної задачі.

Безумовно, коли мова йде про ER-модель, то найчастіше уявляється її графічне зображення. Це не є дивним, адже однією з вимог до вдалого моделювання даних було наявність зрозумілого схематичного зображення. Проте, є багато варіантів утворення такого зображення. В даній роботі буде проілюстрований підхід на основі UML (Unified Modeling Language). Це зумовлено тим, що в сучасному світі комп'ютерних розробок, у будь-якій сфері, починаючи від баз даних і закінчуючи нейронними мережами, UML є уніфікованим стандартом схематичного зображення структури програми. На рисунку нижче показано приклад типової ER діаграми.

Database ER Diagram Example (Crow's Foot)

Lindi H | June 27, 2019

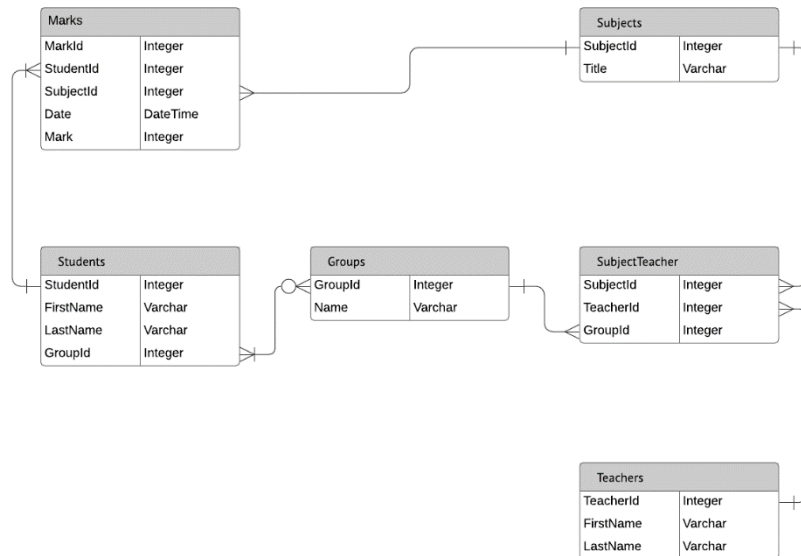


Рисунок 2.1 - Приклад ER діаграми

Базовим поняттям, яке використовується в ER-моделюванні, являється тип сутності. Він являє собою групу об'єктів, що мають однакові властивості та незалежне існування. Насправді, строго формального визначення типу сутності не існує, саме тому, моделювання є достатньо суб'єктивним і кожен розробник, який його проводить, може ідентифікувати сутності відмінно від іншого розробника.

Також дуже важливим поняттям ER-моделювання є поняття екземпляру сутності – об'єкту, що є однозначно ідентифікованим об'єктом, який відноситься до сутності деякого визначеного типу [9].

Графічне зображення сутностей в ER-моделюванні відбувається за допомогою прямокутника, який має в собі ім'я сутності.



Рисунок 2.2 - Позначення сутності

Таким чином, за допомогою наявного підходу можна схематично описати модель зв'язків між таблицями бази даних, тому даний підхід є дуже важливим та актуальним.

2.3 Описання таблиць та зв'язків між ними за допомогою механізму ключів

Після наведення описання ER-діаграм, стає очевидним, що зв'язки між відношеннями бази даних та їх розуміння є фундаментально важливим для вдалого проектування бази даних. Зв'язки між таблицями бази даних, тобто між сутностями, які утворені з предметної області, формуються за допомогою зовнішніх ключів (foreign keys). Тема зовнішніх ключів і ключів загалом вже була дотично розглянута у цій роботі, але для подальшого розуміння концепції організації баз даних з предметної області, необхідно надати точні формулювання ключам та навести їх види. Це дуже важливо, тому що розуміння даної теми є необхідним для вдалого перенесення предметної області у відношення бази даних.

Отже, ключ - це спосіб ідентифікації рядків у таблиці. За допомогою ключів можна пов'язувати записи у різних таблицях між собою [10]. Також можна сказати, що ключ – це колонка або набір колонок, тобто фактично атрибутів відношення, яка унікально ідентифікує записи таблиці.

Можна виділити наступні типи ключів (варто зазначити, що деякі з цих типів не є самостійними, а лише доповнюють інші) [11]:

- суперключ (super key) – це тип ключа, який являє собою будь-яку комбінацію полів, що унікально ідентифікує кожен запис таблиці;
- потенційний ключ (candidate key) – це мінімальний суперключ таблиці. Він має являти собою набір полів, які мають відповідати наступним умовам: не повинно бути можливості скоротити потенційний ключ, цей тип ключа не має мати NULL значення, а натомість тільки унікальні значення незалежно від будь-яких змін, що відбуваються в таблиці. Строго кажучи, від суперключа, даний тип відрізняється тільки меншою кількістю полів, окрім випадку найменшого суперключа. Таблиця має включати хоча б один потенційний ключ, але їх може бути і більше, якщо необхідно;
- первинний ключ (primary key) – це потенційний ключ у відношенні реляційної моделі бази даних, який був обраний в якості основного ключа. Цей вид ключа має відповідати обмеженням, які застосовуються для потенційного ключа, а також наступним умовам: він обов'язково має мати значення, яке він не має часто змінювати. Але бажано, щоб потенційний ключ ніколи не змінював своє значення. Також умовою є його унікальність для кожного запису таблиці на весь час;
- альтернативний ключ (alternative key) – це потенційний ключ, який не є первинним;
- сурогатний ключ – це термін, яким називають службове поле (атрибут), яке додане у відношення реляційної бази даних до полів, які вже визначені. Єдиною метою цього типу ключів є функціонування в якості первинного ключа. Значення цього типу ключів зазвичай генеруються автоматично. Прикладом такого атрибуту є, наприклад, унікальний ідентифікатор деякого відношення, тобто ID. Сурогатний ключ має ряд достатньо відчутних переваг в порівнянні з первинним ключем, що формується на основі полів таблиці. Дуже важко уявити ситуацію при якій значення сурогатного ключа буде змінено, тому що воно ніяк не залежить від інших даних у таблиці. Якщо ж значення сурогатного ключа усе-таки треба замінити, то зробити це не важко, адже це не буде залежати від інших

полів. Також він є достатньо економним, адже, як правило, задається деяким числом. Це більш виграно у порівнянні зі звичайним первинним ключем, який формується з полів таблиці і є більш важким. Сурогатний ключ гарантує унікальність значно краще ніж первинний, що являє собою композицію звичайних полів, оскільки ці поля можуть змінюватись з плином часу і виникає ризик отримання аномалій;

- зовнішній ключ – це явище, яке трохи відрізняється від інших типів ключів, але також відноситься до них. Під зовнішнім ключем треба розуміти деяку підмножину атрибутів деякого відношення, значення яких співпадають зі значеннями деякого потенційного ключа іншого відношення.

Необхідно сказати, що виділяють два типи ключів – прості і зіставні. Як не складно зрозуміти з визначень: простий ключ складається з одного поля, а зіставний - з декількох атрибутів.

За допомогою концепції ключів можна накладати обмеження на відношення бази даних. Ряд таких обмежень є дуже широким, починаючи від гарантування унікальності записів і закінчуючи встановленням правил формування зв'язків між відношеннями. Механізм ключів, як і механізм тригерів, про які мова буде йти далі, дозволяє значно спростити розробку та проектування бази даних. Етап створення ключів є необхідним під час декомпозиції предметної області у відношення.

Визначення первинних ключів необхідно проводити ще до фізичного створення бази даних. Це один із перших етапів проектування. Від самого початку треба чітко розуміти та визначитися з принципами, за якими будуть обиратися первинні ключі для відношень. Як наведено в інформації раніше, є два підходи у формуванні цього типу ключів. Перший – це формування його з полів, які є у відношенні і несуть деяке смислове навантаження. Такий підхід називається природним і, як на мене, є достатньо неідеальним так як поля таблиці можуть змінитися з часом і це призведе до зміни унікального ідентифікатора

записів відношення. Це є дуже ризикованим варіантом, який може призвести до потенційного виникнення аномалій під час взаємодії з серверною частиною програми, яка буде вимушена підлаштовуватися під зміни в базі даних. Виходячи з даних фактів, а також зі свого практичного комерційного досвіду розробника, я вважаю, що кращим підходом до визначення первинного ключа може слугувати концепція сурогатного ключа. Переваги цього підходу були описані вище, коли надавалося визначення. Але, використовуючи концепцію сурогатного ключа, дуже важливо правильно обрати спосіб генерації. Адже, невдало згенерований ключ, який, наприклад, не буде гарантувати унікальності, може призвести до виникнення помилок при функціонуванні програмного комплексу в майбутньому.

Можна виділити декілька варіантів генерації сурогатного ключа. Перший підхід – це автоматично збільшувати ID при додаванні нових записів у таблицю. Єдиним мінусом цього підходу є складність контролювання, до якого числа треба додавати значення. Цю складність можна проілюструвати наступним прикладом: припустимо, що є таблиця з даними студентів. Там 150 записів, останній з яких – це студент з ID, що рівне 150. Якщо з цієї таблиці буде видалено 100 записів і останнім стане 50-й студент, то постає питання: який ID має бути у випадку додавання нового запису, 51 чи 151? На мою думку, значно кращий варіант це встановити число 151, тому що це виключає можливість випадково почати взаємодіяти з даними для студента, який був 51-м до видалення з таблиці (так, це значить, що десь дані в таблиці були сформовані неправильно, адже при видаленні запису, має видалятися в взаємодія з ним, проте, варто виключати ризик повністю). В цілому, запам'ятовувати останнє присвоєне ID і додавати значення для нього в разі створення нового запису – достатньо надійний підхід, який підтримується у сучасних СУБД. Ще одним підходом є генерація випадкового числа. Але такий підхід потребує, щоб в якості типу даних для такого числа виступав тип, який має дуже широкий діапазон допустимих значень. Наприклад, long.

Також існує підхід генерації деякого випадкового строкового значення. Це дозволяє ще ймовірніше виключити можливість випадкового співпадіння ключа, який було згенеровано. Кожен з цих підходів має свої переваги і недоліки, а тому має право на життя, але фактом є те, що первинний ключ – це дуже важлива частина таблиці і, відповідно, всієї бази даних. Тому його формування є відповідальною задачею.

Ще одним механізмом за допомогою якого можна контролювати цілісність даних і зв'язки між ними, є механізм зовнішніх ключів. За допомогою цього типу ключів можна чітко регламентувати правила зв'язку між атрибутами різних відношень. Це дозволяє досягти кращої структури даних та автоматичної перевірки коректності у випадку деяких небажаних дій. Зовнішні ключі не просто обмежують зв'язки між відношеннями, вони, фактично і являють собою ці зв'язки, сформовані за певними правилами. За їх допомогою можна ілюструвати і програмувати такі типи зв'язку як «один до одного», «багато до одного», «багато до багатьох». Кожен з цих типів зв'язку реалізується по-різному. Для зв'язку «один до одного» та «багато до одного» достатньо створити зовнішній ключ без необхідності створення нових таблиць. Тоді як для відображення зв'язку «багато до багатьох» треба створити таблицю-посередника, що буде відображати зв'язки. Проте, зробити це все неможливо без механізму зовнішніх ключів.

Визначившись з необхідністю зовнішніх ключів для створення бази даних з предметної області, необхідно також розуміти, які проблеми можуть виникати при відображенні зв'язку між відношеннями. Традиційно виділяють наступний ряд проблем [12]:

- аномалія видалення. Виникає коли з головної таблиці видаляється запис. В цьому випадку, зовнішній ключ з видаленої таблиці продовжує посилатися на запис, якого не існує. Для усунення даної проблеми можна встановити одне з наступних обмежень: встановити каскадне оновлення (тобто механізм, який автоматично буде видаляти запис із залежної таблиці при

видаленні запису з головної таблиці), або, у випадку, видалення запису з головної таблиці, встановлювати NULL в якості значення, для пов'язаного значення залежної таблиці (щоб реалізувати цей спосіб, необхідно допустити можливість для атрибуту зовнішнього ключа мати значення NULL);

- аномалія вставки. Виникає, коли запис додається в залежну таблицю. В цьому випадку зовнішній ключ із залежної таблиці не відповідає первинному ключу жодного з записів головної таблиці. Для вирішення цієї проблеми, при додаванні в залежну таблицю нових даних, атрибут, що являє собою зовнішній ключ, повинен допускати NULL в якості значення. Таким чином, якщо буде додаватися запис, який не має зв'язку з головною таблицею, то в значенням зовнішнього ключа буде стояти NULL;
- аномалія оновлення. При цій аномалії кілька записів таблиці можуть містити дані, які належать одному і тому ж об'єкту. При зміні даних в одному записі вони можуть викликати протиріччя з даними з іншого запису. Для усунення аномалії оновлення можна застосувати підхід нормалізації, теоретичний фундамент якого було розглянуто у минулому розділі.

Вирішення кожної з проблем, що були описані вище, є надзвичайно важливим для успішної розробки комплексу і вдалого формування бази даних з предметної області. В разі, якщо допустити якусь з цих проблем, то гарантовано в майбутньому буде виникати безліч помилок, які буде складно відловити і побічних ефектів. Таким чином, значно вигідніше витратити більше часу на вдале продумування структури обмежень для зовнішніх ключів кожного відношення, ніж потім на виправлення великої кількості помилок. Більш того, сучасні СУБД, такі як MySQL надають ряд чудових інструментів, що дозволяють легко встановити необхідні обмеження автоматично. Наприклад, при створенні зовнішнього ключа в MySQL (як і в інших СУБД) можна обрати дію, яка буде відбуватися при видаленні чи додаванні даних у головну таблицю. Серед таких дій є наступні:

- CASCADE: самостійно змінює або видаляє записів у залежній таблиці при видаленні або зміні пов'язаних записів в головній таблиці;
- SET NULL: при оновленні або видаленні запису з головної таблиці встановлює для зовнішнього ключа значення NULL;
- RESTRICT: забороняє зміну або видалення записів в головній таблиці при наявності пов'язаних записів у залежній таблиці;
- NO ACTION: має дію, що цілком аналогічна до RESTRICT;
- SET DEFAULT: при видаленні пов'язаного запису з основної таблиці цей тип обмежень ставить для зовнішнього ключа значення за замовчуванням, яке визначається за допомогою атрибуту DEFAULT. Не підтримується InnoDB.

2.4 Накладання обмежень з предметної області на реляційну модель за допомогою механізму тригерів

Ключі, в тому числі зовнішні вирішують задачу забезпечення зв'язку між відношеннями бази даних та в деякому сенсі забезпечують базову перевірку даних у таблицях на правильність. Але, одного механізму ключів недостатньо для досягнення максимально можливого ступеня цілісності даних. Постає ряд проблема, яку не можна вирішити за допомогою ключів або типізації полів у відношеннях. Необхідно мати логічну цілісність даних. Цілісність даних є основною складовою інформаційної безпеки. У своєму найширшому використанні поняття «цілісність даних» стосується точності та послідовності даних, що зберігаються в базі даних [16]. Для забезпечення цілісності даних важливо вирішити як мінімум дві проблеми.

- а) Необхідно перевірити коректність даних. За допомогою ключів та типізації ми можемо досягти наступного: деяке поле може бути деякого типу та мати значення, що відповідає деякому полю в іншій таблиці.

Таким чином, гарантовано, що, наприклад, поле ID не буде null, та, в разі накладання на нього обмеження через зовнішній ключ, дане поле буде відповідати полю з іншого відношення. Але, як бути, наприклад, з полем «Ціна» таблиці «Автомобіль», якщо це поле не обмежене зовнішнім ключем? Безумовно, обравши тип DECIMAL для цього атрибуту, ми гарантуємо деякий рівень коректності введених даних. Але ми все ще не маємо гарантії того, що це буде додаткове число, або число, що є нижчим за поріг найвищої ціни, що встановлена бізнесом і вимогами замовника. Таким чином, виникає необхідність гарантувати логічну коректність введених даних для цього атрибуту. Наприклад, що ціна не може бути від'ємною, або більшою за деяке число. Варто зазначити, що перевірка даних є необхідною умовою для досягнення цілісності даних [17];

- б) Друга проблема полягає в досягненні автоматизації оновлення взаємозалежних даних у системі. У якомусь сенсі, автоматичного оновлення даних можна досягти зовнішнім ключем. Дійсно, атрибути відношення можуть оновлюватись автоматично в разі оновлення відповідних атрибутів з іншого відношення, що пов'язані зовнішнім ключем. Проте, як бути з деякими вимогами бізнесу, які не можуть бути покриті за допомогою механізму зовнішніх ключів? Для прикладу розглянемо базу даних пологового будинку. Покладемо, що кожна пацієнтка може перебувати в стані 1 (вагітна), або 0 (народила). При народженні дитини її вносять в таблицю «Немовлята». Таким чином, при реєстрації немовляти його мама вже перейшла зі стану вагітності у стан «народила». Але як зафіксувати це в системі? Звичайно, записи в таблиці «Пацієнтки» та «Немовлята» пов'язані зовнішнім ключем. Але ми маємо гарантувати, що при народженні нового малюка, його запис точно буде внесено в реєстраційну книгу. Тобто, при зміні стану матері, має додатися новий запис в таблицю «Немовлята». Ця задача, як і попередня, вирішується за допомогою механізму тригерів.

Тригер – це механізм, що ініціює дію щоразу, коли змінюється рядок або набір рядків. Зміни можуть бути наступні: вставити, оновити або видалити [18].

Тригери допомагають вирішувати проблему забезпечення цілісності даних. Необхідно сказати, що, насправді, ж інші механізми, які вирішують цю задачу. Проте, в багатьох сенсах, тригери є хорошим варіантом, який при правильному підході дає багато можливостей для регуляції поведінки у базі даних.

Традиційно можна виділити наступні три типи тригерів, які підтримуються більшістю сучасних СУБД:

- а) BEFORE. Тригери цього типу виконуються перед зміною запису відношення, на яке цей тригер накладений. Такі тригери використовуються тільки для заборони операції, в разі некоректності даних (можна сигналізувати про помилку, додавши інформативне повідомлення), або для встановлення деякого значення «за замовчуванням» у таблицю, на яку цей тригер накладено. Тригери BEFORE типу не можуть бути використанні для викликання змін у іншій таблиці (на яку їх не накладено). Таким чином, ці тригери вирішують тільки першу з двох проблем, що були описані вище;
- б) AFTER. Тригери цього типу використовуються для проведення деякої дії після зміни запису відношення, на які їх накладено. На відміну, від тригерів типу BEFORE, цей тип може бути використаний не тільки для вирішення першої проблеми (тобто перевірки введених даних на коректність та оновлення даних відношення, на яке накладено тригер), а і для зміни записів у іншій таблиці. Таким чином, за допомогою AFTER механізму можна розкрити механізм тригерів на повну потужність та продукувати автоматичне оновлення записів деякого відношення, або додавання нових рядків, при зміні чи додаванні рядка у іншу таблицю. Безумовно, цей механізм є важливим фактором в досягненні цілісності;

в) **INSTEAD OF**. Цей тип тригерів виконується замість деякої дії, яке виконує користувач. Таким чином, дія користувача не виконується взагалі. На відміну від тригерів інших типів, для цього можливо визначити тільки одну операцію, що може бути виконана. Варто зазначити, що часте використання цього типу є сумнівним з точки зору прозорості дій у системі. Оскільки, вони можуть вводити в оману розробника та провокувати «сторонній ефект», коли дія, яка має деякий сенс, буде фактично виконувати зовсім іншу функцію (по факту, дія не буде виконуватись, але зі сторони це може бути не помітно). Проте, іноді використання таких тригерів є необхідним, але ставитися до цього варто з обережністю.

Додатково, до типів тригерів, існує три типи операцій, які можуть виконувати тригери [19]:

- а) **INSERT**. Ця операція виконується після вставлення нового запису у відношення, на яке накладений тригер;
- б) **UPDATE** . Виконується в разі оновлення запису відношення, на яке накладений тригер;
- в) **DELETE**. Даний тип операцій виконується після видалення запису з відповідного відношення.

Типи тригерів, сумісно з видами операцій, які можуть ними виконуватись, надають широкий простір для виконання різноманітних задач забезпечення цілісності даних. Для прозорості варто сказати, що як і будь-яке рішення, даний механізм має ряд недоліків, які коротко будуть описані в наступному розділі даної роботи. Проте, безумовно, тригери є комфортним та потужним методом вирішення поставлених задач.

Висновки до розділу

В даному розділі було коротко розглянуто основні методи та концепції для перенесення предметної області у відношення бази даних.

Основними підходами у формуванні бази даних з предметної області є висхідний та низхідний, хоча, на практиці, найчастіше, дані підходи комбінуються.

Також важливим пунктом даного розділу було моделювання даних. Наведено основні вимоги до того, щоб модель даних вважалася успішною. Було зазначено, що найкращим способом описання моделі даних є схема відношень і зв'язків між ними, яка задається за допомогою EER діаграми.

Було розглянути механізм ключів, наведено основні їх типи та наведено пояснення необхідності використання даного механізму для перевірки даних на коректність. Також, зазначено важливість зовнішніх ключів для правильного описання правил зв'язків між різноманітними відношеннями бази даних.

В останньому підрозділі описано деякі проблеми, які виникають при перенесенні предметної області в базу даних. А саме – забезпечення коректності даних та автоматичного оновлення логічно пов'язаних даних. Розглянуто механізм, що забезпечує можливість гнучкого вирішення даних задач – тригери. Наведено типи тригерів, що зустрічаються в сучасних СУБД, перераховано особливості кожного з цих типів. Також, наведено типи операцій, які можуть виконувати тригери.

РОЗДІЛ 3: РОЗРОБКА СИСТЕМИ ПІДТРИМКИ ДАНИХ З УПРАВЛІННЯ ГОТЕЛЬНИМ КОМПЛЕКСОМ

3.1 Формування схеми бази даних з предметної області

Після визначення основних підходів до розробки складних систем підтримки розподілених даних та огляду основних механізмів, які дозволяють проектувати такі системи, у минулих розділах, перейдемо тепер до виконання практичної частини дипломної роботи.

Як відомо, першим дуже важливим етапом є формування схеми бази даних з деякої предметної області. Виходячи з теми даної роботи, якою є наступне: «Програмна система для підтримки розподілених даних по управлінню завантаженістю готельного комплексу», можна сказати, що предметною областю є готельний бізнес, а задачею – програмний комплекс з підтримки його функціонування. Тому, після проведення необхідного аналізу за допомогою низхідного проектування, було вирішено сформулювати наступні сутності:

- Користувач. Основна сутність, яка має в собі дані про людину, яка буде жити в готелі. Має атрибути для абстрактного описання людини. Має сурогатним ключем поле ID;
- Кімната. Сутність, яка характеризує собою кімнату, в якій може жити один чи декілька користувачів. Вона має атрибути, необхідні для описання реальної кімнати з предметної області. Має сурогатним ключем поле ID;
- Тип кімнати. Сутність що описує можливі категорії кімнат та діапазон цін за ніч для кожної з них. Має в якості первинного ключа поле ID кімнати;
- Бронювання. На відміну від перших двох сутностей, бронювання є абстрактним, воно характеризує дію користувача по відношенню до кімнати. Має сурогатним ключем поле ID;
- Оплата. Це сутність, яка покликана в деякому сенсі доповнювати Бронювання. Вона має в собі атрибути, необхідні для описання фінансової сторони дії користувача по відношенню до оплати. Має в якості первинного ключа поле ID бронювання;
- Відміна. Сутність, яка створюється в разі відміни бронювання. Має в якості первинного ключа поле ID бронювання.

Варто зазначити, що саме така декомпозиція предметної області у відношення реляційної бази даних є суб'єктивною та не може претендувати на повну вичерпність. Проте, дана модель цілком описує основні сутності і операції для фундаменту автоматизації роботи готельного бізнесу.

Після створення сутностей наступним важливим етапом є задання відповідних зв'язків між ними, які задаються за допомогою зовнішніх ключів. На малюнку нижче можна побачити EER-діаграму схеми, на якій за усіма правилами проілюстровані зв'язки між таблицями.

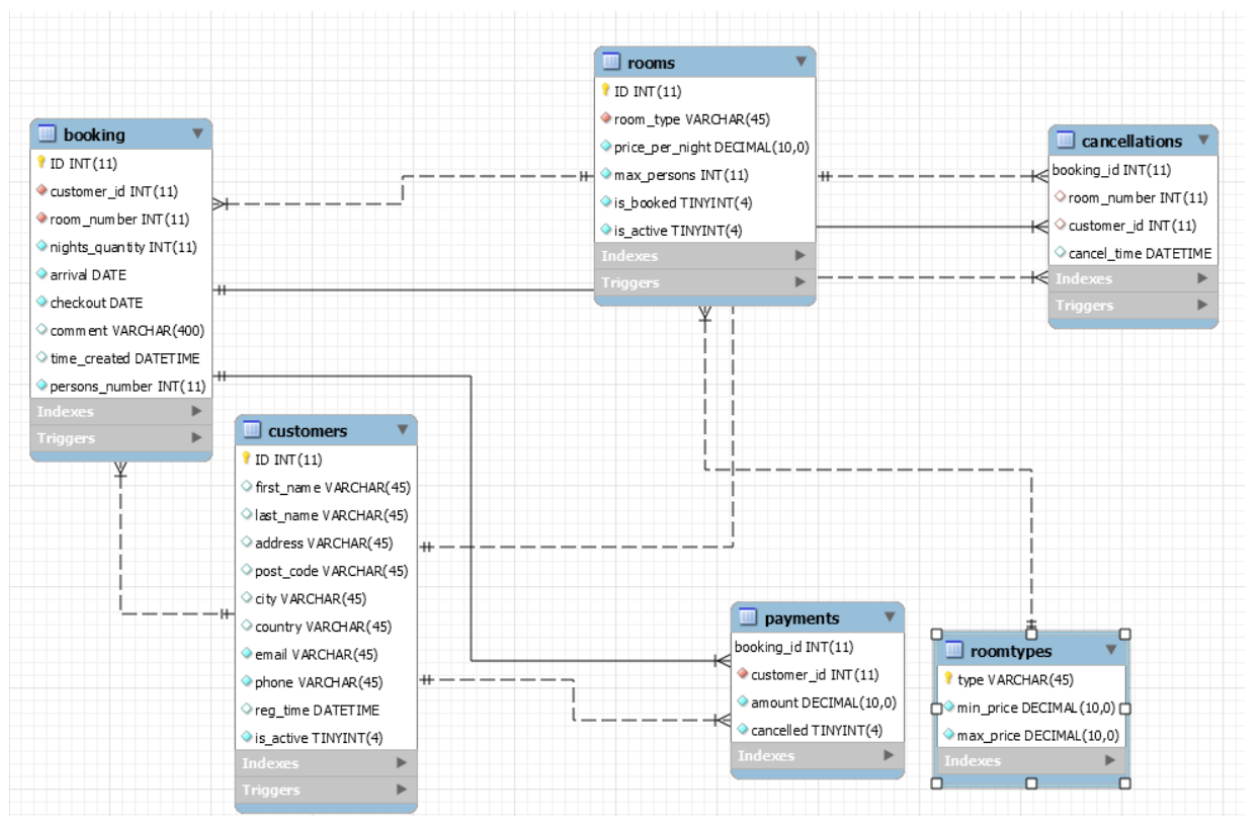


Рисунок 3.1 - EER-діаграма бази даних

Було вирішено використовувати СУБД MySQL для роботи з базою даних, оскільки дана технологія вирізняється своєю простотою та легкістю у використанні. При цьому, вона має увесь набір підтримки необхідних функцій, таких як зовнішні ключі і додаткові опції для них, тригери та інше.

В даній схемі були задіяні наступні зовнішні ключі для описання зв'язку атрибутів між таблицями:

- поле room_type таблиці rooms пов'язано з полем Type таблиці roomtypes;
- поле booking_id таблиці payments пов'язане зовнішнім ключем з полем ID таблиці booking;
- поле customer_id таблиці payments пов'язане зовнішнім ключем з полем ID таблиці booking;
- поле room_number таблиці booking пов'язане зовнішнім ключем з полем ID таблиці rooms;

- поле `customer_id` таблиці `booking` пов'язане зовнішнім ключем з полем `ID` таблиці `customers`;
- поле `booking_id` таблиці `cancellations` пов'язане зовнішнім ключем з полем `ID` таблиці `booking`;

Після проведення аналізу необхідних зв'язків, було досягнуто висновку, що наведеного набору зовнішніх ключів цілком достатньо для встановлення належних обмежень в базі даних. Варто зазначити, що було обрано наступну додаткову умову для зовнішніх ключів: `NO_ACTION`.

3.2 Накладання додаткових обмежень на схему бази даних

Після створення таблиць та описання необхідних зв'язків між ними, ми уже маємо деякий набір мінімальних правил та обмежень для регулювання майбутньої роботи бази. Серед них є: точне визначення типів даних для атрибутів таблиць, механізм первинних ключів, механізм зовнішніх ключів.

Проте, даного набору механізмів недостатньо для задання усіх необхідних правил, які мають контролювати коректність роботи програмного комплексу. Необхідно чітко гарантувати цілісність даних після оновлення таблиць та валідувати нові записи. Після проведення декомпозиції та аналізу, було виявлено наступні основні обмеження, яким має відповідати програмний комплекс для підтримання правильного стану даних:

- кімнату не можна деактивувати, якщо в ній є користувач;
- при створенні бронювання має створитися відповідна оплата;
- в кімнату не можна заселити число людей більше, ніж максимальне;
- не можна деактивувати користувача, якщо на даний момент не витік строк функціонування його бронювання;

- у випадку бронювання кімната стає заблокованою, а користувач, що забронював, не може більше оформлювати бронювання на ці дати;
- дати активних бронювань однієї кімнати не мають перетинатися;
- дати активних бронювань одного користувача не мають перетинатися
- ціна за ніч для номера не має порушувати ціновий діапазон для його категорії;
- у випадку відміни бронювання кімната з цього бронювання має стати вільною для бронювань, а користувач має мати можливість бронювати на ці дати.

Наступним питанням, яке постає після формування необхідних обмежень, є практична реалізація цих обмежень. Є декілька шляхів для виконання цієї задачі. Перший підхід полягає у використанні механізму тригерів за допомогою яких можна налаштувати валідацію та зміну однієї таблиці після зміни в іншій. Ще одним підходом є реалізація кожного з вищезгаданих пунктів у серверній частині програмного комплексу. Наприклад, за допомогою відповідних класів і методів. Ще одним варіантом може бути реалізація і в базі даних і в серверній частині, для максимального страхування. Але, цей підхід є дуже затратним.

Провівши аналіз, було обрано варіант реалізації наявних обмежень через механізм тригерів. Безумовно, цей підхід має ряд мінусів: він менш гнучкий, оскільки такий функціонал більш просто реалізувати в об'єктно-орієнтовному підході, також, сучасні технології, такі як Spring Boot, ще більше спрощують досягнення подібних цілей і дозволяють робити це дуже швидко. Важливим недоліком є те, що тригери роблять логіку програми менш явною. Проте, підхід тригерів забезпечує одну дуже важливу перевагу: значно більшу безпеку транзакцій, особливо коли це стосується задач, коли одна таблиця має змінюватися в результаті змін іншої таблиці. При виконанні таких завдань через об'єктний код зі сторони серверної частини, необхідно зробити дві операції: змінити одну таблицю, а потім, відразу після цього, іншу. Це призводить до того, що одна логічна дія має виконатися за дві операції, тобто постає необхідність

створення транзакції [13]. А, як відомо, з минулих розділів, забезпечення безпеки виконання транзакцій, особливо в паралельному режимі, є нетривіальною задачею, яка потребує багато зусиль. Саме тому, при імплементації програмних комплексів подібного роду постулюється принцип: виконувати як можна менше запитів до бази даних і виконувати всі необхідні операції за найменшу кількість звернень. Але, насправді, як було сказано у минулому розділі даної роботи, сам механізм тригерів також являє собою транзакційний підхід до виконання групи запитів. Тобто, транзакція буде створена що при використанні тригерів, що при множинних запитах зі сторони сервера. Проте, в даному випадку не треба буде встановлювати повторне з'єднання з базою, не буде додаткового звернення, зменшиться кількість запитів. Більш того, транзакції, в ході яких виконуються тригера, контролюються самою СУБД, такою як MySQL в даному випадку чи будь-якою іншою із сучасних. Це дозволить витратити менше ресурсу і зусиль на контролювання процесу транзакцій. Таким чином, хоча тригери мають ряд недоліків, окрім проблем з «прозорістю» також те, що вони можуть загальмувати виконання коду, бо вони буду виконуватись кожен раз після оновлення таблиці [14], даний механізм все одно є прийнятним, особливо для систем невеликого розміру.

Отже, відповідно до кожної з дев'яти задач, описаних вище, було створено тригери, підв'язані під різні дії з відношеннями (наприклад, AFTER_INSERT або інше). Не всі з цих тригерів є вартими особливої уваги, але всі вони надані у додатку А до цієї роботи. Проте, нижче наведемо, найбільш цікаві з них, для ілюстрації можливостей даного механізму.

Більшість тригерів мали достатньо просту реалізацію, але виконання умов 4, 6 і 7 із переліку, наведеного вище, був нетривіальною задачею. Складність виконання кожної з цих задач полягала в тому, щоб правильно працювати з часовими відрізками, що можуть перетинатися. Треба було точно виключити всі неправильні варіанти. Наведемо код тригерів під номером 6 та 7, тобто таких, що

вирішують задачу контролю за перетином дат бронювання для деякої кімнати або користувача:

```
IF (SELECT COUNT(ID) FROM (SELECT ID FROM booking WHERE
((DATEDIFF(new.arrival, booking.arrival) >= 0 AND DATEDIFF(booking.checkout, new.arrival) >= 0)
OR
(DATEDIFF(new.checkout, booking.arrival) >= 0 AND DATEDIFF(booking.checkout, new.checkout) >= 0)
OR
(DATEDIFF(booking.arrival, new.arrival) >= 0 AND DATEDIFF(new.checkout, booking.checkout) >= 0))
AND
(booking.room_number = new.room_number OR booking.customer_id = new.customer_id)
AND
booking.ID != new.ID) as ids
INNER JOIN payments ON
ids.ID = payments.booking_id
WHERE payments.cancelled = 0) > 0 THEN
    signal sqlstate '45000' set message_text = 'Booking timeslots intersection for room';
END IF;
```

Рисунок 3.2 - Перевірка перетину дат бронювання для деякого користувача або кімнати

Як можна побачити, в цьому тригері виключаються всі можливі неправильні перетини дат. Таким чином досягається те, що в разі введення некоректної дати прибуття чи відправлення для користувача, кімнати, або них обох (тобто тої дати, що перетинається з датою хоча б одного невідміненого бронювання) буде відправлена помилка зі статусом 45000 і повідомленням про перетин часових відтинків.

3.3 Реалізація запитів до бази даних

Після того, як з предметної області було сформовано схему реляційної бази даних та після накладення необхідних обмежень для цілісності даних і реалізації механізмів, що виконують ці обмеження, наступним, дуже важливим етапом є програмування необхідних запитів до бази даних, за допомогою SQL. Пізніше, ці запити буде виконувати серверна частина програмного комплексу, написана на мові програмування Java, по користувацькому запиту, що є

загальноприйнятим підходом, який застосовується, наприклад, в корпорації Oracle [15]. Тобто, SQL запити представляють собою основний функціонал системи.

Завдяки проведенню аналізу, перш за все емпіричним способом, заснованим на власному досвіді, було досягнуто висновку, що програмний комплекс, який являє собою систему управління готельним комплексом, має виконувати хоча б 18 основних задач. Список цих задач можна побачити у додатку В, так як нема сенсу перераховувати їх усі. Проте, серед цього списку наявні як і достатньо очевидні запити, які є дуже простими, наприклад, обрання користувача за електронною поштою, виведення списку усіх користувачів, або отримання усіх бронювань деякого користувача, так і достатньо нетривіальні задачі. Таким непростим завданням, яке вимагає об'єднання декількох таблиць та збір інформації з кожної з них, можна вважати, наприклад, виведення даних усіх користувачів, що забронювали номери в деякий період часу, або запит, який показує усі вільні кімнати, які будуть доступні на певні дати. На перший погляд другий запит є дуже простим, але насправді є ризик помилки з вирахуванням перетину часових відрізків. Також, сама по собі проблема полягає в тому, що вільна кімната на деяку дату не обов'язково не буде мати бронювання на цю дату, просто воно може бути відмінене, тому при виконанні цього запиту треба враховувати різні фактори та робити достатньо велику кількість різноманітних перевірок.

Програмну реалізацію усіх запитів можна побачити у додатку Б. В основній частині роботи наведу тільки код запиту під номером 10 та 12, тобто «Показати всі вільні номери на деякі дати» та «Вивести дані усіх користувачів, що забронювали номери деякого типу в деякий період часу». Причина особливої уваги до цих запитів полягає не тільки в тому, що описано в минулому абзаці, а ще і в тому, що є очевидним: в системі адміністрування готельним комплексом такі запити будуть виконуватись дуже часто, тому вони представляють важливий функціонал.

```
SELECT Distinct(rooms.ID), room_type, price_per_night, max_persons, is_active FROM rooms
LEFT JOIN booking ON booking.room_number = rooms.ID
LEFT JOIN payments ON payments.booking_id = booking.ID
WHERE (rooms.is_booked = 0 AND DATEDIFF('2021-06-04', NOW()) >= 0 AND DATEDIFF('2021-11-05', '2021-06-04') >= 0)
OR ((DATEDIFF(booking.arrival, '2021-11-05') > 0 OR DATEDIFF('2021-06-04', booking.checkout) > 0)
AND DATEDIFF('2021-06-04', NOW()) >= 0 AND DATEDIFF('2021-11-05', '2021-06-04') >= 0 AND payments.cancelled = 0)
```

Рисунок 3.3 - Запит на отримання усіх вільних кімнат в деяких період

```
SELECT customers.ID, first_name, last_name, email, post_code, city, country, phone, reg_time, customers.is_active FROM customers
INNER JOIN booking ON booking.customer_id = customers.ID
INNER JOIN rooms ON rooms.ID = booking.room_number
INNER JOIN payments ON payments.booking_id = booking.ID
WHERE ((DATEDIFF(booking.arrival, ':startDate') >= 0) AND (DATEDIFF(':endDate', booking.checkout) >= 0))
AND rooms.room_type = ':roomType' AND payments.cancelled = 0
```

Рисунок 3.4 - Запит на отримання даних усіх користувачів, що забронювали номери деякого типу на деякий період часу

3.4 Розробка серверної частини програмної системи з підтримки даних для управління готельним комплексом

3.4.1 Обрання підходу до програмування

Основною частиною програмного комплексу, який є результатом даної розробки, є база даних, оскільки саме там зберігається інформація, а більшість логіки програми реалізовано за допомогою SQL запитів та механізму тригерів і ключів. Та, незважаючи на це, реалізація серверної частини застосунку є надзвичайно важливим елементом для успішного виконання даної роботи.

Як було описано в минулих розділах, більшість застосунків такого типу являють собою клієнт-серверні веб-додатки. А на серверну частину програми покладається відповідальність в «спілкуванні» з базою даних, тобто отримання та запису даних в неї.

Під час виконання даної роботи, було обрано об'єктно орієнтовний підхід у реалізації програмних сутностей серверної частини застосунку. Враховуючи, що предметна область, що розглядається в даній темі, на цьому рівні конкретності, не вимагає дуже великої кількості сутностей, а зв'язки між ними не є надто заплутаними, можна стверджувати, що об'єктно-орієнтовний підхід містить в собі всі необхідні характеристики для досягнення поставлених цілей.

3.4.2 Засоби для розробки серверної частини програми

Першим, дуже важливим, етапом розробки є обрання мови програмування, якою буде написано код. Враховуючи, що було вирішено розробляти об'єктно-орієнтований код, необхідно обрати мову, яка надасть максимальні можливості в роботі з ООП. При цьому, дана мова має бути достатньо простою, безпечною та мати достатньо інструментів для роботи з базами даних та веб-сферою. Виходячи з вищенаведених вимог та особистого опиту комерційного використання, було вирішено обрати мову програмування Java. Наведемо декілька причин, чому було обрано саме її.

- Java є об'єктно орієнтовною мовою за своєю сутністю. Кожна програма на цій мові є об'єктно-орієнтовною [20];
- має потужні інструменти для роботи з веб-клієнтами та базами даних;
- надає багато можливостей для досягнення гнучкості програмної архітектури. Це досягається за рахунок наявності багатьох потужних елементів, які можна використати для побудови класів та інтерфейсів [21].

Після обрання мови, було вирішено, що основною бібліотекою, яка допоможе у розробці, буде обрано Spring. Дана бібліотека являє собою набір інструментів, що полегшують роботу з об'єктно-орієнтовними системами, дозволяють спростити комунікацію з базою даних та налагодження зв'язку в веб-клієнтами. Фундаментальною проблемою, яку вирішує цей набір інструментів, є

полегшення програмування на мові Java [22]. Особливо, в ході виконання дипломної роботи було використано наступні модулі Spring:

- Spring Core. Основний модуль, який полегшує створення та управління програмними сутностями;
- Spring JPA. Модуль, який дозволяє встановлювати зв'язок з базою даних;
- Spring WEB. Модуль, який дозволяє розробляти web-орієнтовні застосунки.

3.4.3 Архітектура та сутності програмного комплексу

Перед описанням архітектури, яка була реалізована у додатку, для початку треба сказати про програмні сутності, які було вирішено розробити. Власне, ці сутності абсолютно дублюють відношення, які є у схемі бази даних. Ними є:

- Кімната;
- Користувач;
- Бронювання;
- Оплата;
- Відміна.

Визначення саме таких сутностей дозволяє вирішувати проблему об'єктно-реляційного відображення значно простіше, ніж якщо б об'єкти відрізнялися від реляційних відношень.

Після визначення програмних сутностей перейдемо до описання архітектури, яка була реалізована у додатку. Безумовно, розробка застосунків даного виду є типовою задачею у сучасному програмуванні. Механізми дій, які дозволяють вирішувати повторювану задачу, незалежно від деяких нюансів, називаються шаблонами, або патернами розробки. Будь-який патерн описує завдання, яке знову і знову виникає в роботі, а також принцип його рішення, причому таким

чином, що це рішення можна потім використовувати багато разів, нічого не змінюючи [23]. Саме тому, для розробки поточної задачі, було вирішено використати один із патернів - модифікований шаблон розробки MVC (а саме Controller-Service-Repository модифікація). За допомогою цього шаблону архітектуру серверного коду можна розбити на три частини:

- Controller. Приймає та віддає дані клієнту;
- Service. Обробляє дані від контролера, проводить деякі логічні дії з ними і віддає їх на шар нижче – репозиторію. Після отримання відповіді від репозиторію передає дані на контролер. Саме наявність сервісної частини є важливою відмінністю даної модифікації шаблону від звичайного MVC. Присутність цього шару дає більшу безпеку, гнучкість та розбиття коду [24];
- Repository. Отримує дані від сервісу та надсилає їх до бази даних. Отримує відповідь від бази та надсилає її сервісу.

Для програми підтримки даних з управління готельним комплексом було створену наступну архітектуру на базі обраного вище шаблону.

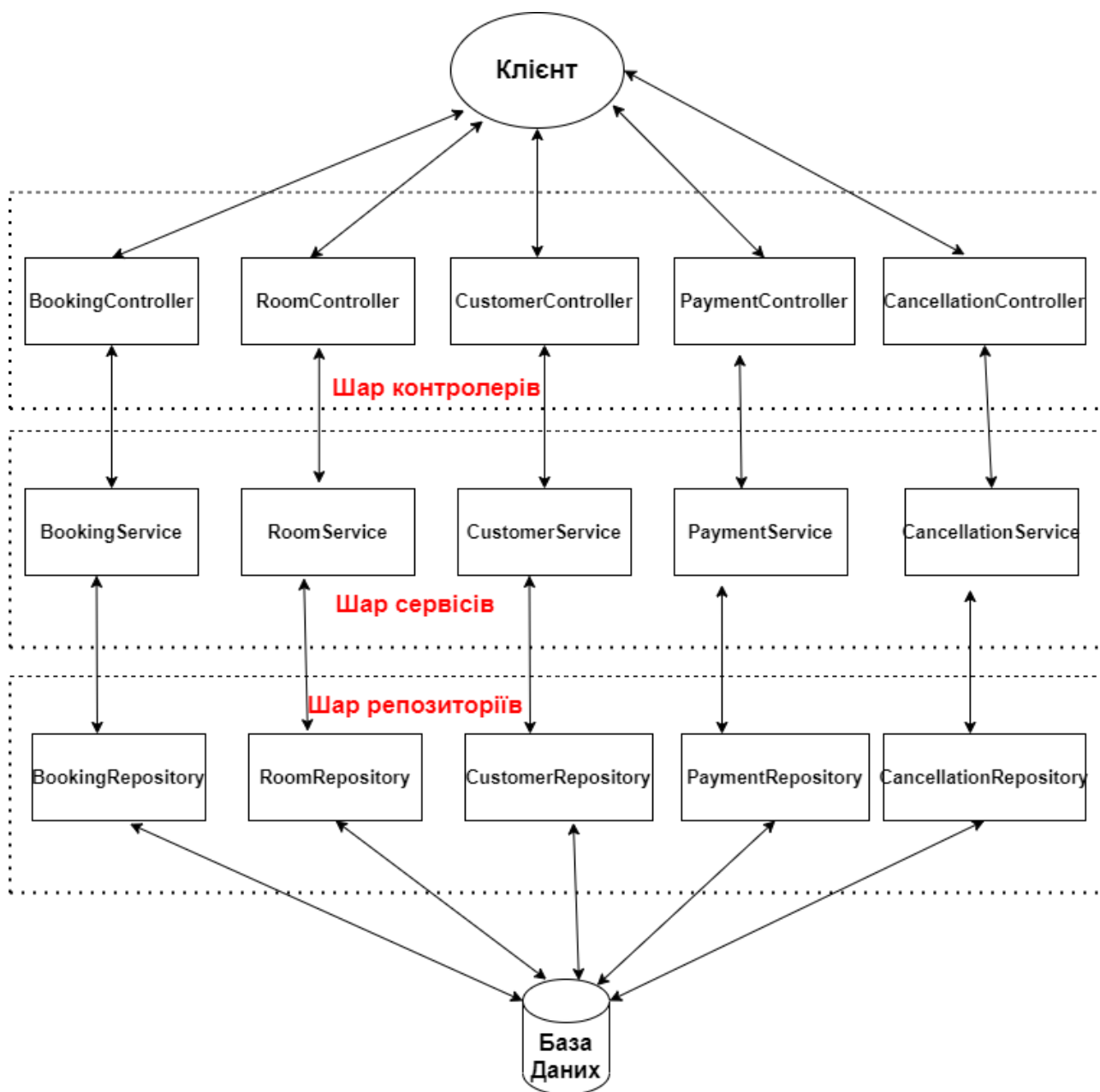


Рисунок 3.5 - Архітектура програмного комплексу

В кожному з класів, що наведені на діаграмі, присутні методи (тобто функції), які вирішують поставлені задачі, що відносяться до конкретної сутності, яка покривається даним функціоналом. Необхідно зазначити, що завдяки реалізації основної частини обмежень через механізм тригерів, була досягнута простота зв'язків між об'єктами серверного коду. Як видно на схемі, кожен компонент взаємодіє лише з представниками, що відносяться до тієї самої сутності, просто на різних рівнях ієрархії. Це значно спрощує розуміння коду та його гнучкість.

3.4.4 Клієнтська частина та інтерфейс програми

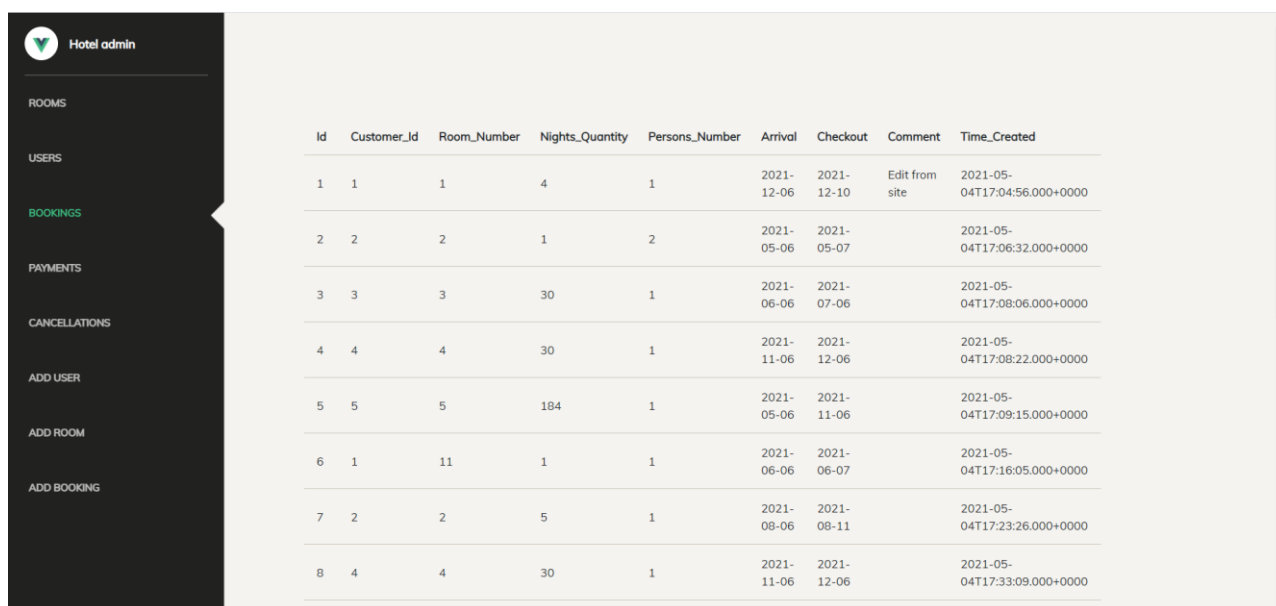
Функціонал системи, яка є ціллю даної роботи, було розроблено у серверній частині та у базі даних. Фактично, ці дві частини являють собою готову систему. Але, кінцевий користувач навряд зможе з легкістю користуватися такою системою. Саме тому, для успішного виконання роботи необхідно розробити клієнтську частину, яка буде містити інтерфейс та код, що відправляє запити на серверну частину.

В даній роботі було вирішено розробити код клієнтської частини за допомогою VueJs. VueJs – це JavaScript бібліотека, для створення користувацьких інтерфейсів (views) на основі базових моделей даних [25]. Це сучасна технологія, яка дозволяє створювати потужні та швидкі додатки та з легкістю встановлювати комунікацію з сервером.

Було створено інтерфейс, який являє собою дошку адміністратора сайту. Цей інтерфейс дозволяє користувачеві отримувати всю необхідну інформацію з бази даних, а також продукувати необхідні зміни, за допомогою інтуїтивно зрозумілих функцій.

Наведемо приклади інтерфейсу для роботи з деякими функціями програмного продукту, який було створено. Всі функції наводити нема сенсу, бо вони повністю відповідають SQL запитам, що описані у додатку Б.

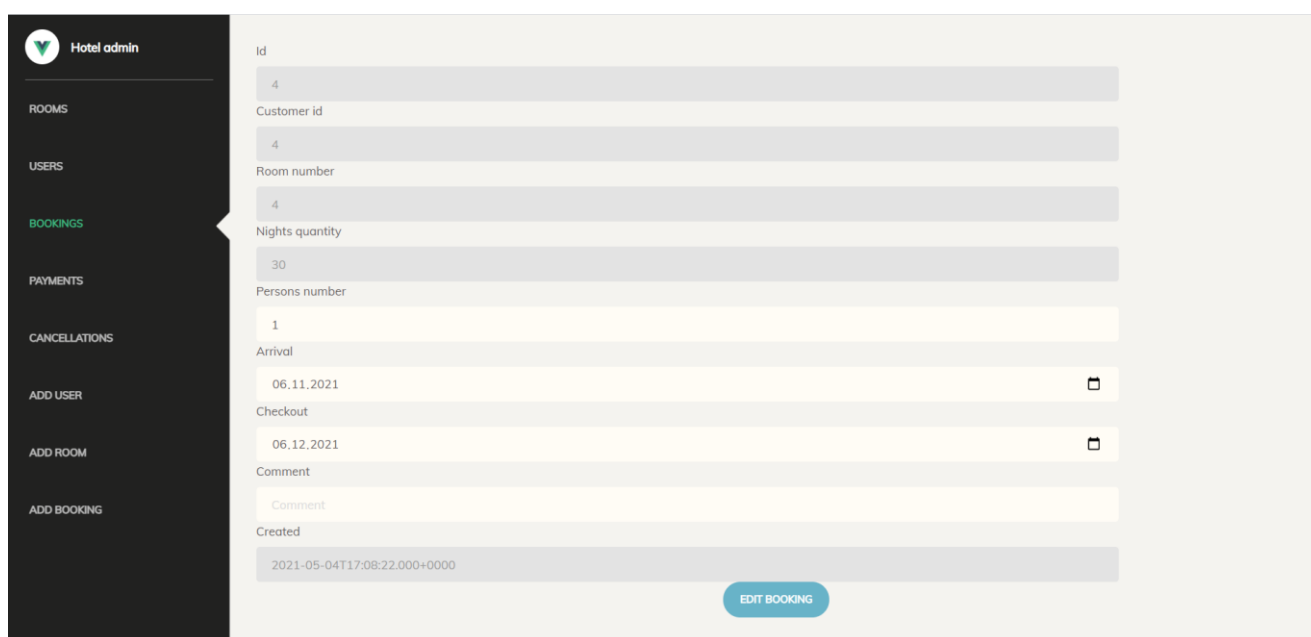
Програмний продукт надає можливість працювати з усіма сутностями, для прикладу, таблиця бронювань.



Id	Customer_Id	Room_Number	Nights_Quantity	Persons_Number	Arrival	Checkout	Comment	Time_Created
1	1	1	4	1	2021-12-06	2021-12-10	Edit from site	2021-05-04T17:04:56.000+0000
2	2	2	1	2	2021-05-06	2021-05-07		2021-05-04T17:06:32.000+0000
3	3	3	30	1	2021-06-06	2021-07-06		2021-05-04T17:08:06.000+0000
4	4	4	30	1	2021-11-06	2021-12-06		2021-05-04T17:08:22.000+0000
5	5	5	184	1	2021-05-06	2021-11-06		2021-05-04T17:09:15.000+0000
6	1	11	1	1	2021-06-06	2021-06-07		2021-05-04T17:16:05.000+0000
7	2	2	5	1	2021-08-06	2021-08-11		2021-05-04T17:23:26.000+0000
8	4	4	30	1	2021-11-06	2021-12-06		2021-05-04T17:33:09.000+0000

Рисунок 3.6 - Вікно переліку бронювань

При натисненні на рядок таблиці відбувається перехід на сторінку конкретного бронювання, де його можна відредагувати



Id	4
Customer id	4
Room number	4
Nights quantity	30
Persons number	1
Arrival	06.11.2021
Checkout	06.12.2021
Comment	Comment
Created	2021-05-04T17:08:22.000+0000

EDIT BOOKING

Рисунок 3.7 - Вікно редагування бронювання

Редагування відбувається відповідно до вимог коректності даних. Також можна побачити, що є поля, які неможливо редагувати. Це забезпечує логічну цілісність даних та обмежує потенціал шкідливих дій.

У випадку, якщо буде введено некоректні дані, користувачеві буде показано повідомлення про помилку.

Hotel admin

ROOMS

USERS

BOOKINGS

PAYMENTS

CANCELLATIONS

ADD USER

ADD ROOM

ADD BOOKING

We're sorry, we're not able to update booking now, please try back later

Id

4

Customer id

4

Room number

4

Nights quantity

30

Persons number

1

Arrival

06.11.2021

Checkout

06.12.2021

Comment

Comment

Created

2021-05-04T17:08:22.000+0000

EDIT BOOKING

Рисунок 3.8 - Вікно редагування бронювання з сигналом про помилку

Hotel admin

ROOMS

USERS

BOOKINGS

PAYMENTS

CANCELLATIONS

ADD USER

ADD ROOM

ADD BOOKING

Id

Id

Customer id

Customer id

Room number

Room number

Persons number

Persons number

Arrival

ДД.ММ.РРРР

Checkout

ДД.ММ.РРРР

Comment

Comment

ADD BOOKING

Рисунок 3.9 - Вікно створення нового бронювання

The screenshot displays the 'Hotel admin' web application. On the left is a dark sidebar with a 'Hotel admin' header and a list of navigation items: ROOMS, USERS, BOOKINGS, PAYMENTS, CANCELLATIONS, ADD USER (highlighted in green), ADD ROOM, and ADD BOOKING. The main content area is a light gray form titled 'ADD USER'. The form contains the following fields: 'Id' (with a sub-field 'id'), 'Registration time', 'Email' (with a sub-field 'Email'), 'First Name' (with a sub-field 'First Name'), 'Last Name' (with a sub-field 'Last Name'), 'Address' (with a sub-field 'Home Address'), 'City' (with a sub-field 'City'), 'Country' (with a sub-field 'Country'), 'Postal Code' (with a sub-field 'ZIP Code'), and 'Phone' (with a sub-field 'Phone'). A blue 'ADD USER' button is located at the bottom right of the form.

Рисунок 3.10 - Вікно додавання нового користувача

Висновки до розділу

В даному розділі було продемонстровано результати розробки програмного комплексу з підтримки розподілених даних по управлінню готельним комплексом та описано особливості програми та логіку її створення.

Було проведено декомпозицію предметної області у відношення бази даних та створено відповідні таблиці. Наведено інтуїтивно зрозумілу схему (EER діаграму), яка описує усі зв'язки та типові особливості кожної з сутностей предметної області задачі, що розглядається у даній роботі.

Основна частина логіки застосунку та зв'язку між відношеннями реалізована у базі даних за допомогою ключів, тригерів та, власне, SQL запитів. Було наведено деякі з запитів та тригерів, а саме найбільш цікаві та нетривіальні та надане пояснення обрання саме таких функцій для системи. Розглянуто переваги механізму тригерів в порівнянні з перевіркою коректності у коді серверної

частини та обґрунтовано використання саме такого підходу, описано особливості його реалізації у цільовому додатку на основі СУБД MySQL.

У цьому розділі було описано архітектуру серверної частини цільового додатку. Архітектура базується на шаблоні проектування Controller-Service-Repository. Завдяки цьому патерну та інструментам, що доступні у мові Java та бібліотеці Spring вдалося досягти гнучкого коду, не перевантаженого зайвими зв'язками між компонентами програми. Логічні шари коду взаємодіють між собою за простою схемою, яка також була наведено графічно, щоб спростити сприйняття.

Було коротко описано підхід до реалізації клієнтської частини коду та наведено приклади роботи інтерфейсу програми та деяких функцій, що виконує клієнтська частина коду.

РОЗДІЛ 4: ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

4.1 Постановка завдання

Розробка програмного забезпечення з підтримки розподілених даних по управлінню готельним комплексом вимагає розробки бази даних, серверної та клієнтської частин програми. В даному розділі проводиться функціонально-вартісний аналіз варіантів та підходів до реалізації компонентів системи.

4.2 Обґрунтування функцій та параметрів дослідження

Враховуючи основні компоненти програми, тобто ті частини, які являють собою модулі, що відповідають за успішне функціонування, наведемо основні функції, що впливають на вартість розробки програмного комплексу, що вирішує поставлену задачу. Основні функціями є:

F_1 – бібліотека для реалізації клієнтської частини програми: а) VueJs; б) React; в) JSP;

F_2 – мова програмування серверної частини: а) Java; б) C#; в) Python;

F_3 – СУБД для реалізації бази даних: а) PostgreSQL; б) MySQL; в) Microsoft SQL.

Наведемо морфологічну карту системи, що продемонструє усі можливі комбінації для реалізації програмних функцій.

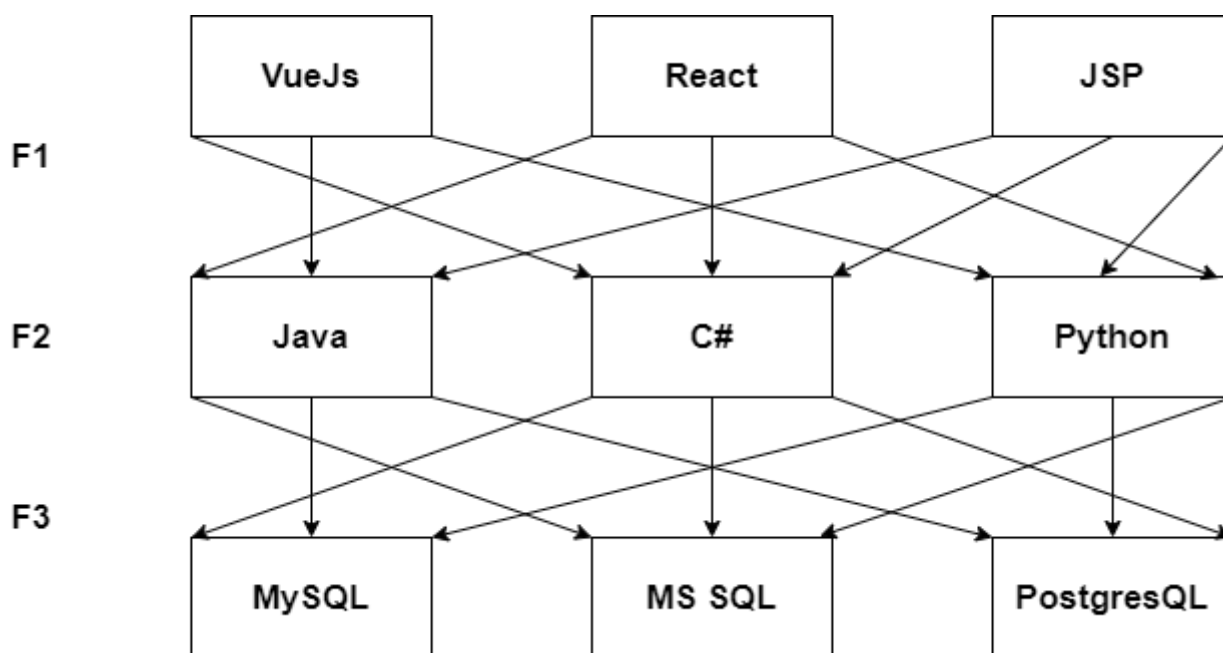


Рисунок 4.1 - Морфологічна карта системи

На основі морфологічної карти можна скласти позитивно-негативну матрицю.

Таблиця 4.1 - Позитивно-негативна матриця

Основна функція	Варіант реалізації	Переваги	Недоліки
F1	А)	Швидкість роботи	Менше функціоналу доступно
	Б)	Повільна робота	Складність освоєння
	В)	Простота розробки	Застарілість технології
F2	А)	Велика кількість допоміжних інструментів	Складність розробки
	Б)	Велика кількість бібліотек для розробки	Відсутність кросплатформності
	В)	Простота розробки	Не оптимально для даної задачі
F3	А)	Швидкість роботи	Середня складність розробки
	Б)	Простота розробки ПЗ	Велика кількість помилок в коді СУБД
	В)	Великий набір інструментів	Висока складність розробки

4.3 Обґрунтування системи параметрів

Після визначення основних функцій можна визначити основні параметри програмного комплексу, які надалі будуть використані для обчислення коефіцієнта технічного рівня. Будемо використовувати наступні параметри для характеристики системи:

X1 – час на вивчення функцій JavaScript бібліотеки, год.;

X2 – необхідний час для ознайомлення з механізмами мови програмування, год.;

X3 – об'єм пам'яті, що витрачається під час роботи програми, Мб;

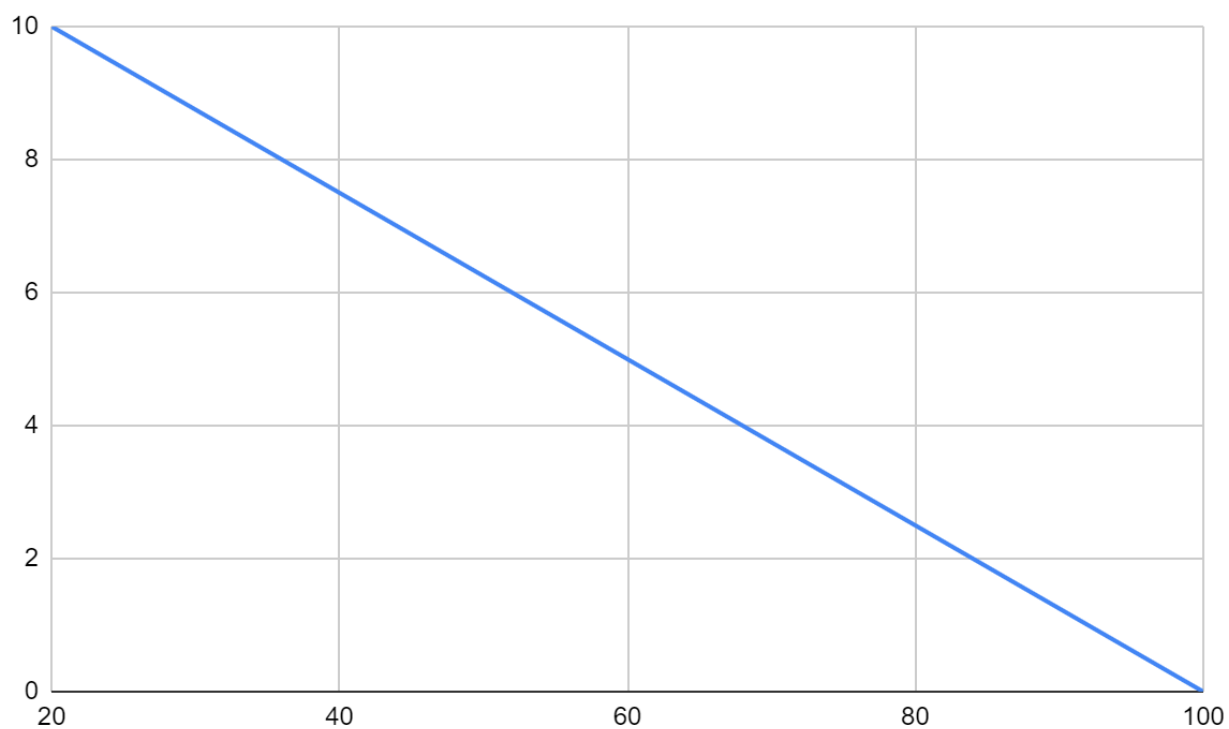
X4 – швидкодія виконання запитів до бази даних, Мс.

Для F1 можна застосувати X1, для F2 відповідають параметри X2 та X3, а параметр X4 застосовується до F3.

Гірші, середні та кращі параметри обрано за допомогою вимог замовника та загальноприйнятих технічних понять.

Таблиця 4.2 - Основні параметри дослідження

Назва Параметра	Умовні позначення	Одиниці виміру	Значення параметра		
			гірші	середні	кращі
Час на вивчення функцій JavaScript бібліотеки	X1	год	60	40	20
Необхідний час для ознайомлення з механізмами мови програмування	X2	год	100	60	20
Об'єм пам'яті, що витрачається під час роботи програми	X3	мб	2048	1024	512



Швидкодія виконання запитів до бази даних	X4	мс	1000	300	20
--	----	----	------	-----	----

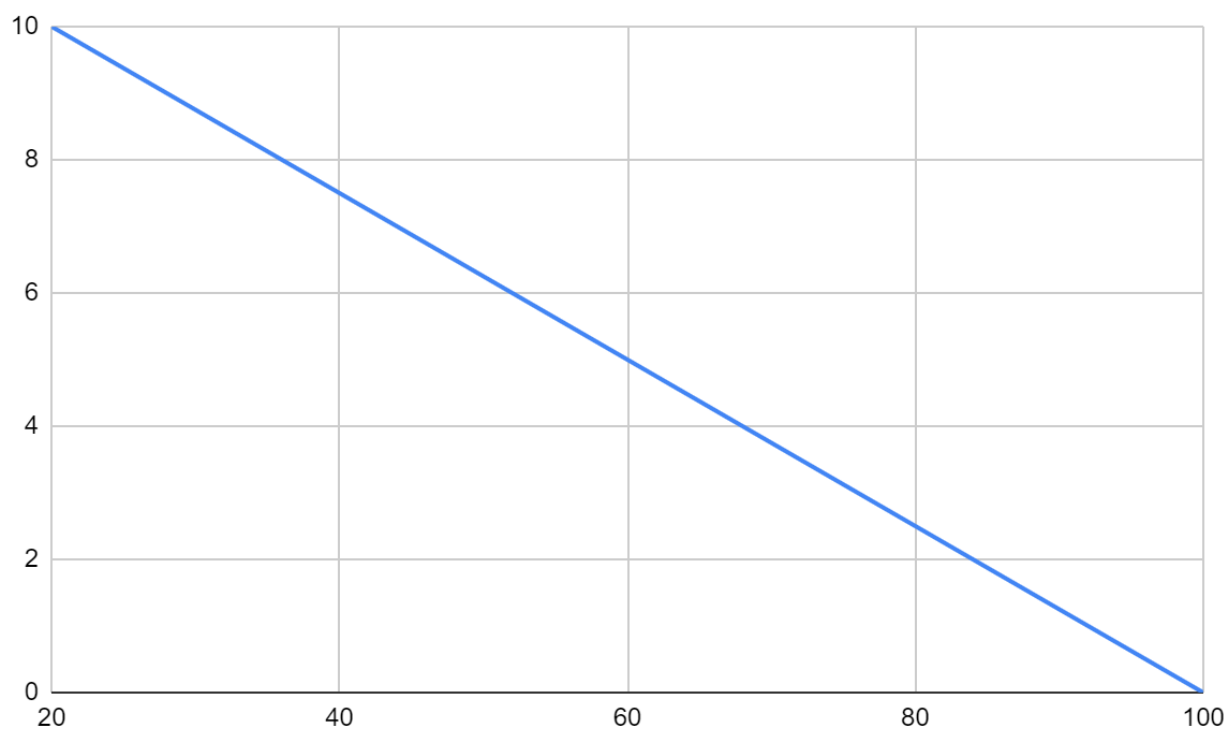


Рисунок 4.2 - Графічне зображення характеристик X1

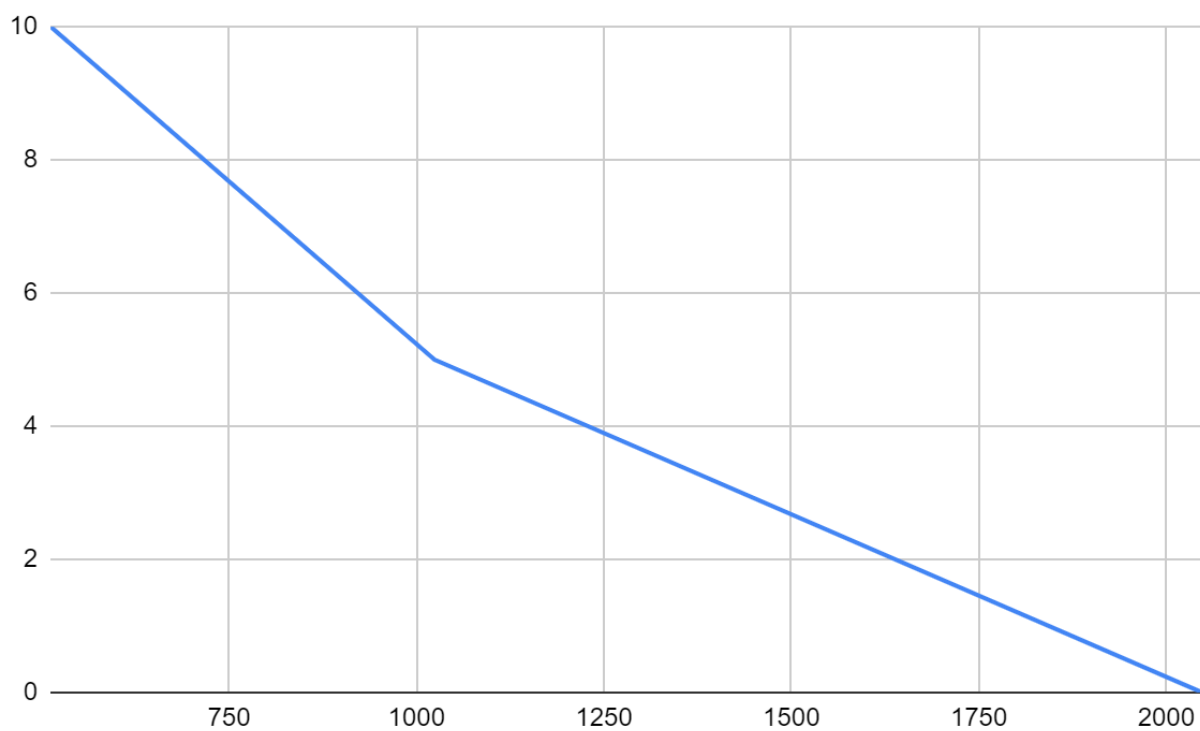


Рисунок 4.3 - Графічне зображення характеристик X2

Рисунок 4.4 - Графічне зображення характеристик X3

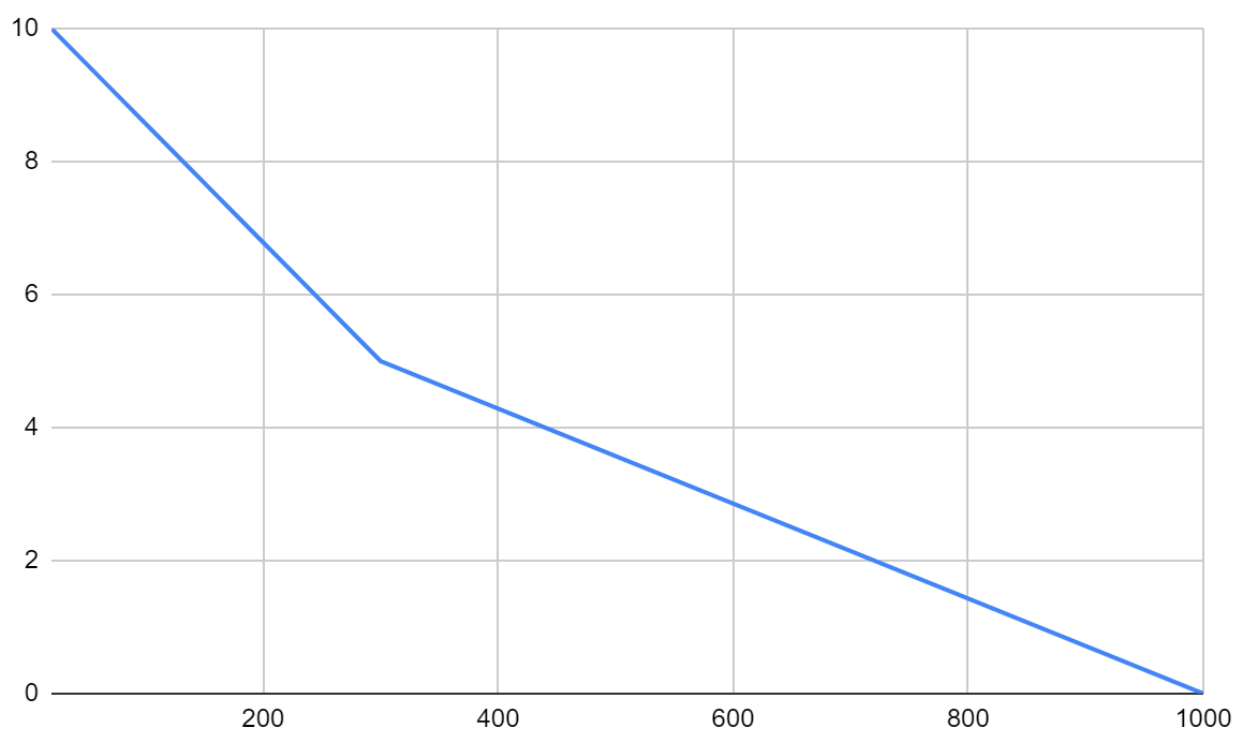


Рисунок 4.5 - Графічне зображення характеристик X4

4.4 Експертне оцінювання важливості параметрів

Важливість кожного з параметрів було визначено завдяки порівняння по парам на основі ранжування експертами та порівняння параметрів. Результати оцінювання експертів наведені у таблиці нижче.

Таблиця 4.3 - Результати експертного ранжування параметрів

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхилення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
$X1$	Час на вивчення функцій JavaScript бібліотеки	Год.	2	2	2	1	1	2	3	15	-4,5	6,25
$X2$	Необхідний час для ознайомлення з механізмами мови програмування	Год.	3	2	3	3	4	3	2	17	2,5	0,25
$X3$	Об'єм пам'яті, що витрачається під час роботи програми	Мб	2	2	1	2	1	1	2	12	-8,5	72,25
$X4$	Швидкодія виконання запитів до бази даних	Сек.	3	4	4	4	4	4	3	26	8,5	72,25
	Разом		10	10	10	10	10	10	10	70	0	171

Наведемо підрахунок коефіцієнту узгодженості за наступною формулою:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 53}{7^2(4^3 - 4)} \approx 0,698 > W_k = 0,67.$$

Значення порахованого коефіцієнту вище за нормативне, тому дані оцінки можна використовувати для проведення подальших досліджень. Для визначення вагомості проведемо попарне порівняння параметрів за допомогою оцінок експертів.

Таблиця 4.4 - Попарне порівняння параметрів

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	<	=	<	<	<	<	>	<	0,5
X1 і X3	=	=	>	<	=	>	>	>	1,5
X1 і X4	<	<	<	<	<	<	=	<	0,5
X2 і X3	>	=	>	>	>	>	=	>	1,5
X2 і X4	=	<	<	<	=	<	<	<	0,5
X3 і X4	<	<	<	<	<	<	>	<	0,5

Для кожного параметра зробимо розрахунок вагомості K_{bi} за наступними формулами:

$$K_{bi} = \frac{b_i}{\sum_{i=1}^n b_i}, \text{ де } b_i = \sum_{j=1}^N a_{ij}.$$

Відносні оцінки розраховуються ітеративно поки наступні значення не різниця значень між ітераціями не стане достатньо малою. На другому і наступних кроках оцінки вираховуються за формулою:

$$K_{bi} = \frac{b'_i}{\sum_{i=1}^n b'_i}, \text{ де } b'_i = \sum_{j=1}^N a_{ij} b_j.$$

Таблиця 4.5 - Розрахунок вагомості параметрів

Параметри x_i	Параметри x_j				Перша ітер.		Друга ітер.	
	X1	X2	X3	X4	b_i	K_{bi}	b_i^1	K_{bi}^1
X1	1,0	0,5	1,5	0,5	3,5	0.21875	12.25	0.208
X2	1,5	1,0	1,5	0,5	4,5	0.28125	16.25	0.2754

X3	0,5	0,5	1,0	0,5	2,5	0.15625	9.25	0.1567
X4	1,5	1,5	1,5	1,0	5,5	0.34375	21.25	0.3601
Всього:					16	1	59	1

З таблиці вище видно, що різниця значень коефіцієнтів на ітераціях не більше 5%, тому достатньо двох ітерацій.

4.5 Аналіз варіантів реалізації функцій програмного комплексу

Проведемо визначення рівнів якості розробки кожного з варіантів функцій.

Коефіцієнт технічного рівня для деякого варіанта реалізації програмного комплексу з підтримки даних можна розрахувати за формулою:

$$K_K(j) = \sum_{i=1}^n K_{\delta i,j} B_{i,j},$$

Таблиця 4.6 - Показники рівня якості для параметрів основних функцій

Основні функції	Варіант реалізації	Параметри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F_1	а)	X_1	20	10	0,208	2,08
F_2	а)	X_2	20	10	0,2754	2,754
		X_3	768	7.5	0,1567	1,175
	б)	X_2	40	5	0,2754	1,377
		X_3	1280	4	0,1567	0,6268
F_3	а)	X_4	100	9	0,3601	3,2409

За даними з таблиці 4.6 за наступною формулою:

$$K_K = K_{\text{Ty}}[F_{1k}] + K_{\text{Ty}}[F_{2k}] + \dots + K_{\text{Ty}}[F_{zk}],$$

визначимо рівень якості кожного з наведених вище варіантів:

$$K_{K1} = 2,08 + 2,754 + 1,175 + 3,2409 = 9,2499$$

$$K_{K2} = 2,08 + 1,377 + 0,6268 + 3,2409 = 7,3247$$

Виходячи з розрахунків можна зробити висновок, що кращим є перший варіант, так як він має кращий коефіцієнт технічного рівня.

4.6 Економічний аналіз варіантів реалізації програмного комплексу

Щоб визначити вартість розробки програмного продукту проведемо розрахунок трудомісткості. Кожен з варіантів реалізації включає в себе наступні задачі:

- а) проектування бази даних;
- б) розробка серверної частини програми;
- в) розробка клієнтської частини програми.

Задачі 2 та 3 віднесемо до 3 групи складності, задачу 1 – до 1 групи складності.

Для задач 2 та 3, враховуючи ступінь новизни Б та характерні норми розробки, трудомісткість дорівнює: $T_p = 84$ людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: $K_{\text{п}} = 0.9$. Коефіцієнт, що показує складність контролю вихідної та вхідної інформації для кожного із завдань рівний $K_{\text{ск}} = 1$. Так як при розробці завдань використовуються стандартні технології та бібліотеки, то $K_{\text{ст}} = 0.8$.

Таким чином, трудомісткість програмування другого і третього завдань дорівнює:

$$T_1 = 62 * 0.9 * 0.8 = 60.48 \text{ людино-днів.}$$

Для першої задачі, виходячи із норм часу для ступеня новизни А та групи складності 1, трудомісткість дорівнює: $T_p = 94$ людино-дня, $K_{\Pi} = 1.7$, $K_{\text{СК}} = 1$, $K_{\text{СТ}} = 0.8$. Тоді:

$$T_2 = 94 * 1.7 * 0.8 = 127.84 \text{ людино-дня.}$$

Найбільш трудомістким є завдання 2. Далі покладемо, що робочий день складає 8 годин, а тиждень містить п'ять робочих днів. В розробці беруть участь три розробника: перший фулстек-розробник початкового рівня з окладом 14000 грн, фулстек-розробник старшого рівня з окладом 71000 грн та тестувальник з окладом 18000 грн. Визначимо середню заробітну плату за годину:

$$C_{\text{ч}} = \frac{14000 + 71000 + 18000}{3 * 21 * 8} = 204,37$$

Заробітна розробників за варіантами становить:

$$C_{\text{ЗП}} = 204,37 * 8 * 60,48 = 98\,882,38$$

$$C_{\text{ЗП}} = 204,37 * 8 * 127,84 = 209\,013,2$$

Відрахування на соціальний внесок становить 22%:

$$C_{\text{ВІД}} = 98\,882,38 * 0,22 = 21\,754,12$$

$$C_{\text{ВІД}} = 209\,013,29 * 0,22 = 45\,982,92$$

Визначимо витрати на оплату однієї машино-години. Так як одна машина обслуговує молодшого фулстек-розробника з окладом 14000 грн, старшого фулстек-розробника з окладом 71000 грн та тестувальника з окладом 18000 грн з коефіцієнтом зайнятості 0.6, то для трьох машин отримаємо:

$$C_T = 12 * 14000 * 0,6 + 12 * 71000 * 0,6 + 12 * 18000 * 0,6 = 741\,600 \text{ (грн)}$$

Враховуючи додаткову заробітну плату 40%

$$C_{ЗП} = 741\,600 * (1 + 0,4) = 1\,038\,240 \text{ (грн)}$$

Відрахування на соціальне страхування 22%

$$C_{ВІД} = 458\,640 * 0,22 = 228\,412,80 \text{ (грн)}$$

Проведемо амортизаційні підрахунки (амортизація 25%, вартість ЕОМ 28000 грн)

$$C_A = K_{ТМ} * K_A * C_{ПР} = 1,15 * 0,25 * 28000 = 8050 \text{ (грн)}$$

Визначимо витрати на профілактику і ремонт:

$$C_P = K_{ТМ} * C_{ПР} * K_P = 1,15 * 28000 * 0,05 = 1610 \text{ (грн)}$$

Визначимо ефективний годинний фонд часу ПК за рік:

$$T_{ЕФ} = (365 - 104 - 10 - 16) * 8 * 0,9 = 1692 \text{ год}$$

Розрахуємо витрати на електроенергію

$$C_{ЕЛ} = 1692 * 0,6 * 0,9 * 3,51 = 3\,207,02 \text{ грн}$$

Накладні витрати наступні:

$$C_H = 28000 * 0,67 = 18\,760 \text{ (грн)}$$

Таким чином можна визначити експлуатаційні витрати:

$$C_{\text{ЕКС}} = 1\,038\,240 + 228\,412,80 + 8050 + 1610 + 3\,207,02 + 18\,760 = 1\,298\,279,82 \text{ (грн)}$$

Собівартість однієї машино-години буде:

$$C_{\text{М-Г}} = \frac{1\,298\,279,82}{1692} = 767,30 \text{ грн/год.}$$

Так як усі роботи, що пов'язані з реалізацією програмного комплексу проводяться на ЕОМ, то витрати на оплату машинного часу в залежності від обраного варіанту, складають:

$$C_{\text{М}} = 767,30 * 8 * 60,48 = 371\,250,43 \text{ грн.}$$

$$C_{\text{М}} = 767,30 * 8 * 127,84 = 784\,733,06 \text{ грн.}$$

Накладні витрати складають 67% від заробітної плати:

$$C_{\text{Н}} = 371\,250,43 * 0,67 = 248\,737,79 \text{ (грн)}$$

$$C_{\text{Н}} = 784\,733,06 * 0,67 = 525\,771,18 \text{ (грн)}$$

Отже, вартість розробки програмного продукту за варіантами становить:

$$C_{\text{ПП}} = 98\,882,38 + 21\,754,12 + 371\,250,43 + 248\,737,79 = 740\,624,72 \text{ (грн)}$$

$$C_{\text{ПП}} = 209\,013,29 + 45\,982,92 + 784\,733,06 + 525\,771,18 = 1\,565\,500,45 \text{ (грн).}$$

4.7 Вибір кращого варіанта ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня:

$$K_{\text{ТЕР}1} = \frac{9,2499}{740\,624,72} = 0,0000124$$

$$K_{\text{ТЕР}2} = \frac{7,3247}{1\,565\,500,45} = 0,0000046$$

Як видно, більш ефективним є перший варіант розробки програми.

Таким чином, виходячи з проведених досліджень, можна сказати, що перший варіант розробки є більш оптимальним, так як має вищі якісно-економічні показники. Його коефіцієнт техніко-економічного рівня складає 0,0000124, в той час як цей коефіцієнт для другого варіанту складає 0,0000046.

Розробка цього варіанту передбачає реалізацію наступного функціоналу як:

- а) розробка веб-інтерфейсу додатку за допомогою VueJs;
- б) розробка серверної частини на мові Java;
- в) проектування БД у СУБД MySQL.

Цей варіант виконання програмного продукту дає користувачу зручний інтерфейс, хороший функціонал і швидкодію.

Висновки до розділу

В якості висновку можна сказати, що у даному розділі було проведено повний функціонально-вартісний аналіз реалізації програмного продукту з підтримки розподілених даних з управління програмним комплексом. Було проведене визначення основних функцій програмного комплексу. Вираховано параметри, що характеризують програмний продукт. Проведено ранжування параметрів експертами та аналіз якості варіантів реалізації функціоналу.

Виходячи з проведеного дослідження можна зробити висновок що оптимальним варіантом є розробка клієнтської частини за допомогою JavaScript

бібліотеки VueJs, серверної частини – за допомогою мови програмування Java, проектування бази даних на основі реляційного підходу в СУБД MySQL.

ВИСНОВКИ ЗА РОБОТОЮ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

Розробка програмних комплексів з підтримки даних є актуальною задачею, яка потребує серйозного ставлення. В даній роботі були розглянуті основні підходи до проектування такого роду систем.

Важливим етапом розробки програмного продукту є проектування бази даних з предметної області. Було визначено основні підходи до проектування та розглянуто найбільш поширені моделі даних. Для практичної реалізації була обрана реляційна модель даних, оскільки вона є перевіреною часом, простою у використанні та має міцну теоретичну основу у вигляді реляційної алгебри.

Наступним етапом розробки була реалізація серверної частини коду. В ході даної роботи увагу було сконцентровано на перевагах клієнт-серверної моделі та розглянуто її особливості.

В якості парадигми програмування було розглянуто ООП, як найкращий варіант реалізації програмного коду серверної частини, проведено аналіз її переваг та обрано Java, як мову програмування серверної частини. Оскільки ця мова має широкий спектр об'єктно-орієнтовних рішень та інструментів у розробці.

Результатом роботи став програмний комплекс, який містить реляційну базу даних, серверну частину та клієнтський код, реалізований за допомогою VueJs.

В якості подальших досліджень можна розглянути реалізацію нового функціоналу системи, такого як диференціювання відвідувачів за віком, проведення оплати окремо від бронювання, можливість завантажити звітність у файл. Також варто дослідити перспективи «хмарних» технологій для відповідності більш сучасним вимогам.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бейли Л. Изучаем SQL. СПб.: Питер, 2012. 198 с.
2. McCreary D., Kelly A. Making Sense of NoSQL: A guide for managers and the rest of us. Manning Publications, 2013. 86 p.
3. Кузнецов С. Д. Основы баз данных. Издание 2-е. Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007. 34 с.
4. Date C.J. What First Normal Form Really Means. Date on database: Writings 2000—2006, Apress, 2006. 45 p.
5. Чеппл М. Базові знання з нормалізації баз дани. URL: <https://www.lifewire.com/database-normalization-basics-1019735> (дата звернення: 11.05.2021).
6. Kent R., William N. A Simple Guide to Five Normal Forms in Relational Database Theory. Communications of the ACM, 1983. 120—125 pp.
7. Дейт К. Дж. Введение в системы баз данных. Изд. 8-е. Москва: Вильямс, 2006. 437 с.
8. Коннолли Т., Бегг К. Базы данных. Проектирование, реализация и сопровождение. Теория и практика. Изд. 3-е. — Москва: Вильямс, 2003. 340 - 344 с.
9. ER diagrams vs. EER diagrams: What's the difference? URL: <https://cacoo.com/blog/er-diagrams-vs-eer-diagrams-whats-the-difference/> (дата звернення: 11.05.2021).
10. Реляционные базы данных. Внешние ключи и связи. URL: <https://metanit.com/sql/tutorial/1.2.php> (дата звернення: 12.05.2021).
11. Database Keys types. URL: http://rdbms.opengrass.net/2_Database%20Design/2.1_TermsOfReference/2.1.2_Keys.html (дата звернення: 12.05.2021).

12. Watt A. Chapter 10 to ER modeling
URL: <https://opentextbc.ca/dbdesign01/chapter/chapter-10-er-modelling/>
(дата звернення: 13.05.2021).
13. Gray C., Jim D. The Transaction Concept: Virtues and Limitations. 7th International Conference on Very Large Databases, 1981. 144—154 p.
14. Триггеры. Преимущества и недостатки. URL:
https://studopedia.su/18_145394_preimushchestva-i-nedostatki-triggerov.html (дата звернення: 16.05.2021).
15. Server-Side Programming: PL/SQL and Java. URL:
<https://docs.oracle.com/database/121/CNCPT/srvrside.htm#CNCPT036>
(дата звернення: 16.05.2021).
16. Teeling M. What is Data Integrity? Learn How to Ensure Database Data Integrity via Checks, Tests, & Best Practices. URL:
<https://www.veracode.com/blog/2012/05/what-is-data-integrity>
(дата звернення: 24.05.2021).
17. Brook C. What is Data Integrity? Definition, Best Practices & More. URL:
<https://digitalguardian.com/blog/what-data-integrity-data-protection-101>
(дата звернення: 24.05.2021).
18. Molinaro A. SQL Cookbook. O'Reilly Media, Inc., 2005. 538 p.
19. Kline K., Kline D. SQL in a nutshell. First edition. O'Reilly Media, Inc., 2001. 37 p.
20. Шилдт Г. Java. Полное руководство. Изд. 10-е. Москва: Диалектика, 2018. 57 с.
21. Блох Д. Java. Эффективное программирование. Изд. 3-е. Москва: Диалектика, 2019. 109 с.
22. Уоллс К. Spring в действии. Москва: ДМК Пресс, 2015. 32 с.
23. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001. 16 с.
24. Alam K. Why to use Service Layer in Spring MVC. URL:

<https://medium.com/stackavenue/why-to-use-service-layer-in-spring-mvc-5f4fc52643c0> (дата звернення: 25.05.2021)

25. Simplified JS jargon. VueJs. URL:

http://jargon.js.org/_glossary/VUEJS.md (дата звернення: 25.05.2021).

Додаток А

Ілюстративний матеріал доповіді

Програмна система для підтримки розподілених даних по управлінню завантаженістю готельного комплексу

Виконав студент 4 курсу

групи КА-74

Поліщук А.Б.

Дипломний керівник: Яковлева А.П.

Опис роботи

- Об'єкт дослідження: програмна система для підтримки розподілених даних по управлінню завантаженістю готельного комплексу
- Предмет дослідження: підхід до розробки програмного комплексу з підтримки даних
- Мета роботи: реалізувати програмний комплекс по управлінню готелем, що допоможе бізнесу розвиватися

Постановка задачі

- Розглянути існуючі рішення і основні концепції в проектуванні систем для підтримки розподілених даних
- Розглянути концепції та методи проектування бази даних з предметної області
- Розробити програмний продукт, що вирішує задачу підтримки даних з управління готельним комплексом
- Продемонструвати результати роботи програмного забезпечення

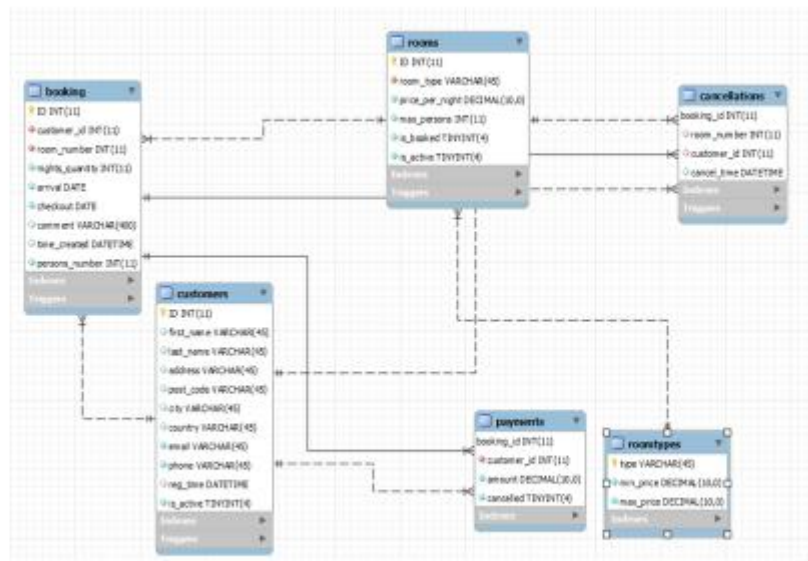
Технології, що були використані у роботі



Spring Boot



ЕЕR-діаграма бази даних



Накладання додаткових обмежень на схему бази даних

Після проведення декомпозиції та аналізу, було виявлено наступні основні обмеження, яким має відповідати програмний комплекс для підтримання правильного стану даних:

- 1. Кімнату не можна деактивувати, якщо в ній є користувач
- 2. При створенні бронювання має створитися відповідна оплата
- 3. В кімнату не можна заселити число людей більше, ніж максимальне
- 4. Не можна деактивувати користувача, якщо на даний момент не витік строк функціонування його бронювання

Накладання додаткових обмежень на схему бази даних

- 5. У випадку бронювання кімната стає заблокованою, а користувач, що забронював, не може більше оформлювати бронювання на ці дати
- 6. Дати активних бронювань однієї кімнати не мають перетинатися
- 7. Дати активних бронювань одного користувача не мають перетинатися
- 8. Ціна за ніч для номера не має порушувати ціновий діапазон для його категорії
- 9. У випадку відміни бронювання кімната з цього бронювання має стати вільною для бронювань, а користувач має мати можливість бронювати на ці дати.

Перевірка перетину дат бронювання для деякого користувача або кімнати

```
IF (SELECT COUNT(ID) FROM (SELECT ID FROM booking WHERE
((DATEDIFF(new.arrival, booking.arrival) >= 0 AND DATEDIFF(booking.checkout, new.arrival) >= 0)
OR
(DATEDIFF(new.checkout, booking.arrival) >= 0 AND DATEDIFF(booking.checkout, new.checkout) >= 0)
OR
(DATEDIFF(booking.arrival, new.arrival) >= 0 AND DATEDIFF(new.checkout, booking.checkout) >= 0))
AND
(booking.room_number = new.room_number OR booking.customer_id = new.customer_id)
AND
booking.ID != new.ID) as ids
INNER JOIN payments ON
ids.ID = payments.booking_id
WHERE payments.cancelled = 0) > 0 THEN
    signal sqlstate '45000' set message_text = 'Booking timeslots intersection for room';
END IF;
```

Реалізація запитів до бази даних

```
SELECT Distinct(rooms.ID), room_type, price_per_night, max_persons, is_active FROM rooms
LEFT JOIN booking ON booking.room_number = rooms.ID
LEFT JOIN payments ON payments.booking_id = booking.ID
WHERE (rooms.is_booked = 0 AND DATEDIFF('2021-06-04', NOW()) >= 0 AND DATEDIFF('2021-11-05', '2021-06-04') >= 0)
OR ((DATEDIFF(booking.arrival, '2021-11-05') > 0 OR DATEDIFF('2021-06-04', booking.checkout) > 0)
AND DATEDIFF('2021-06-04', NOW()) >= 0 AND DATEDIFF('2021-11-05', '2021-06-04') >= 0 AND payments.cancelled = 0)
```

Запит на отримання усіх вільних кімнат в деяких період (дати в цьому прикладі є тестовими, в реальній програмі вони автоматично підставлені)

Реалізація запитів до бази даних

```
SELECT customers.ID, first_name, last_name, email, post_code, city, country, phone, reg_time, customers.is_active FROM customers
INNER JOIN booking ON booking.customer_id = customers.ID
INNER JOIN rooms ON rooms.ID = booking.room_number
INNER JOIN payments ON payments.booking_id = booking.ID
WHERE ((DATEDIFF(booking.arrival, 'startDate') >= 0) AND (DATEDIFF('endDate', booking.checkout) >= 0))
AND rooms.room_type = 'roomType' AND payments.cancelled = 0
```

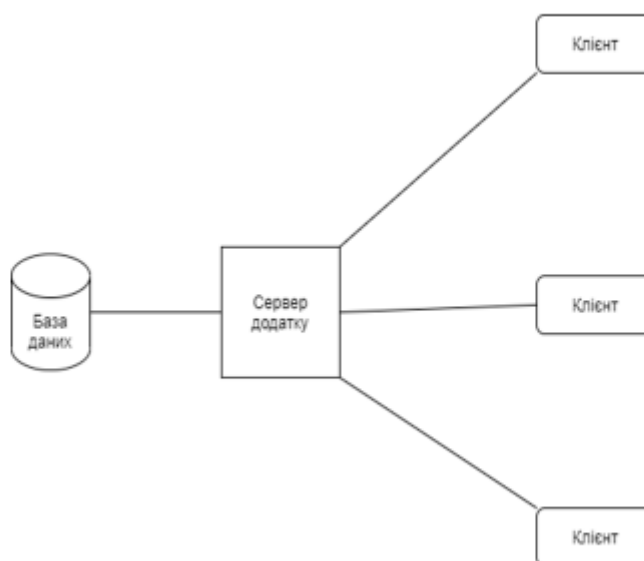
Запит на отримання даних усіх користувачів, що забронювали номери деякого типу на деякий період часу

Сутності програмного комплексу

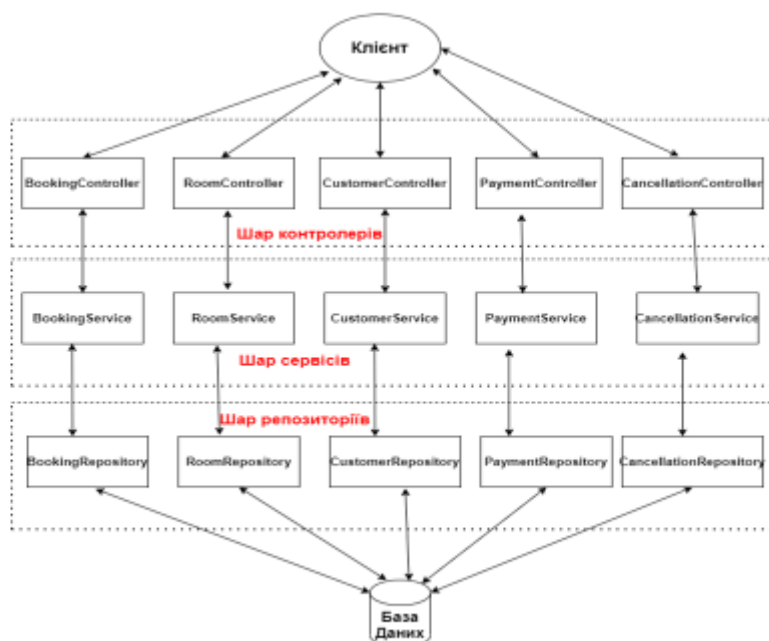
Перед описанням архітектури, яка була реалізована у додатку, для початку треба сказати про програмні сутності, які було вирішено розробити. Власне, ці сутності абсолютно дублюють відношення, які є у схемі бази даних. Ними є:

- Кімната
- Користувач
- Бронювання
- Оплата
- Відміна
- Тип кімнати

Клієнт-серверна архітектура програмних комплексів



Архітектура програмного комплексу



Інтерфейс програми

The screenshot shows the program's interface. On the left is a sidebar menu with options: ROOMS, USERS, BOOKINGS (highlighted in green), PAYMENTS, CANCELLATIONS, ADD USER, ADD ROOM, and ADD BOOKING. The main area displays a table of bookings.

ID	Customer ID	Room Number	Nights/Quantity	Person Number	Start	Check-out	Comment	Time Created
1	1	1	4	1	2021-12-08	2021-12-10	Edit from site	2021-05-04T17:04:58.000+0000
2	2	2	1	2	2021-09-08	2021-09-07		2021-05-04T17:08:32.000+0000
3	3	3	30	3	2021-09-08	2021-07-06		2021-05-04T17:08:38.000+0000
4	4	4	30	3	2021-11-08	2021-12-06		2021-05-04T17:08:22.000+0000
5	5	5	104	1	2021-05-06	2021-11-06		2021-05-04T17:08:15.000+0000
6	1	11	1	1	2021-06-06	2021-06-07		2021-05-04T17:08:05.000+0000
7	2	2	5	1	2021-06-06	2021-06-11		2021-05-04T17:07:26.000+0000
8	4	4	30	3	2021-11-06	2021-12-06		2021-05-04T17:07:09.000+0000

Вікно переліку бронювань

Інтерфейс програми

The screenshot shows the 'Edit reservation' form in a web application. On the left is a dark sidebar with a menu containing: 'ROOMS', 'USERS', 'RESERVATIONS' (highlighted in green), 'PAYMENTS', 'CANCELLATIONS', 'ADVISORS', 'ADVISORS', and 'ADVISORS'. The main content area is light gray and contains a form with the following fields: 'id' (value: 1), 'Customer id' (value: 1), 'Room number' (value: 1), 'Night quantity' (value: 1), 'Person number' (value: 1), 'Status' (value: 1), 'Arrival' (value: 01.11.2021), 'Checkin' (value: 01.11.2021), 'Comment' (empty), and 'Created' (value: 2021-09-04T11:08:22.000+0000). At the bottom right of the form is a blue button labeled 'SAVE'.

Вікно редагування бронювання

Інтерфейс програми

This screenshot shows the same 'Edit reservation' form as above, but with an error message displayed at the top in an orange banner: 'NOT A DATE, NOT A TIME TO CHECK IN, CHECK IN DATE MUST BE DATE TIME'. The form fields and the 'SAVE' button are still visible below the message.

Вікно редагування бронювання з сигналом про помилку

Інтерфейс програми

The screenshot displays a web interface for filtering and searching user requests. It includes several sections with filters and action buttons:

- Top Section:** A 'BUDGET' dropdown menu, date range filters for '06.06.2021' and '07.06.2021', and a 'WHO BOOKS IN PERIOD' button.
- Second Section:** Time range filters for '06.06.2021 00:01' and '07.06.2021 23:59', and a 'CANCELLED IN PERIOD' button.
- Third Section:** A search bar and a 'SEARCH BY EMAIL' button.
- Bottom Section:** A 'BUDGET' dropdown, a checkbox for 'Does book should be cancelled', a 'GET WITH BOOK OF TYPE' button, and another checkbox for 'Does book should be cancelled'.
- Footer Section:** A checkbox for 'Is checkout date', a date filter for '06.06.2021', and a 'GET USER WITH DATE' button.

Панель параметризації запитів про користувачів

Висновки та перспективи розвитку

- В ході роботи було реалізовано базу даних, серверну та клієнтську частини програмного комплексу за допомогою реляційного та об'єктно-орієнтованих підходів. Було використано клієнт-серверну модель архітектури програмного комплексу.
- Програмний комплекс містить необхідний набір функцій для управління готельним бізнесом.
- В якості подальших покращень можна реалізувати додаткові функції, такі як: диференціювання відвідувачів за віком, проведення оплати окремо від бронювання, можливість завантажити звітність у файл.

ДОДАТОК Б

Програмний код продукту

Обмеження бази даних:

```
CREATE DEFINER=`root`@`localhost` TRIGGER
`hotel_system`.`booking_BEFORE_INSERT` BEFORE INSERT ON `booking` FOR
EACH ROW

BEGIN
```

```
    IF(SELECT is_active FROM rooms WHERE ID = new.room_number) = 0

        THEN signal sqlstate '45000' set message_text = 'Can`t create booking
for inactiver room';

    END IF;
```

```
    IF(SELECT is_active FROM customers WHERE ID = new.customer_id) = 0

        THEN signal sqlstate '45000' set message_text = 'Can`t create booking
for inactive customer';

    END IF;
```

```
    IF(DATEDIFF(NOW(), new.arrival) > 0) THEN signal sqlstate '45000' set
message_text = 'Arrival should be later than now'; END IF;
```

```
    /** check if checkout later then arrival **/
```

```
    IF DATEDIFF(new.checkout, new.arrival) < 0 THEN
```

signal sqlstate '45000' set message_text = 'Checkout date should be more than arrival';

END IF;

*/** Nighr quantity is a dif between checkout and arrival time **/*

SET new.nights_quantity = IF(DATEDIFF(new.checkout, new.arrival) > 0,DATEDIFF(new.checkout, new.arrival), 1);

*/** Time created**/*

SET new.time_created = NOW();

*/*3 limitation from file*/*

IF new.persons_number > (SELECT max_persons FROM rooms where ID = new.room_number) THEN

signal sqlstate '45000' set message_text = 'Number of persons is more than allowed';

END IF;

/ 6,7 limitations from file */*

IF (SELECT COUNT(ID) FROM (SELECT ID FROM booking WHERE ((DATEDIFF(new.arrival, booking.arrival) >= 0 AND DATEDIFF(booking.checkout, new.arrival) >= 0)

OR

```

(DATEDIFF(new.checkout, booking.arrival) >= 0 AND
DATEDIFF(booking.checkout, new.checkout) >= 0)

OR

(DATEDIFF(booking.arrival, new.arrival) >= 0 AND DATEDIFF(new.checkout,
booking.checkout) >= 0))

AND

(booking.room_number = new.room_number OR booking.customer_id =
new.customer_id)

AND

booking.ID != new.ID) as ids

INNER JOIN payments ON

ids.ID = payments.booking_id

WHERE payments.cancelled = 0) > 0 THEN

        signal sqlstate '45000' set message_text = 'Booking timeslots
intersection for room';

END IF;

END

CREATE DEFINER=`root`@`localhost` TRIGGER
`hotel_system`.`booking_AFTER_INSERT` AFTER INSERT ON `booking` FOR
EACH ROW

BEGIN

        /** set isBooked to true when room is booked**/

        UPDATE rooms set is_booked = 1 WHERE

                ID = new.room_number;

```

```

/* 2 limitation from file */

INSERT INTO payments VALUES(

new.ID,

new.customer_id,

(SELECT price_per_night FROM rooms where ID = new.room_number) *

new.nights_quantity,

0);

END

```

```

CREATE DEFINER=`root`@`localhost` TRIGGER
`hotel_system`.`booking_AFTER_DELETE` AFTER DELETE ON `booking` FOR
EACH ROW

BEGIN

    /** set isBooked to false when room is unbooked**/

    UPDATE rooms set is_booked = 0 WHERE

        ID=OLD.room_number;

END

```

```

CREATE DEFINER=`root`@`localhost` TRIGGER
`hotel_system`.`cancellations_BEFORE_INSERT` BEFORE INSERT ON
`cancellations` FOR EACH ROW

BEGIN

    SET new.cancel_time = NOW();

```

```
SET new.room_number = (SELECT room_number from booking where
booking.ID = new.booking_id);
```

```
SET new.customer_id = (SELECT customer_id from booking where booking.ID =
new.booking_id);
```

```
END
```

```
CREATE DEFINER=`root`@`localhost` TRIGGER
`hotel_system`.`cancellations_AFTER_INSERT` AFTER INSERT ON `cancellations`
FOR EACH ROW
```

```
BEGIN
```

```
/* 9 limitation from file */
```

```
UPDATE payments
```

```
SET cancelled = 1 WHERE payments.booking_id = new.booking_id;
```

```
UPDATE rooms
```

```
SET is_booked = 0 WHERE rooms.ID = new.room_number;
```

```
END
```

```
CREATE DEFINER=`root`@`localhost` TRIGGER
`hotel_system`.`customers_BEFORE_INSERT` BEFORE INSERT ON `customers`
FOR EACH ROW
```

```
BEGIN
```

```
SET new.reg_time = NOW();
```

```
END
```

```

CREATE DEFINER=`root`@`localhost` TRIGGER
`hotel_system`.`customers_BEFORE_UPDATE` BEFORE UPDATE ON `customers`
FOR EACH ROW

BEGIN

    /*4 limitation from file*/

    IF (SELECT count(payments.cancelled) from payments

        INNER JOIN booking

        on booking.ID = payments.booking_id

        where booking.customer_id = booking.customer_id

        and payments.cancelled = 0

        and DATEDIFF(booking.checkout, NOW()) > 0

        and booking.customer_id = new.ID

        and new.is_active = 0

    ) > 0

    THEN

        signal sqlstate '45000' set message_text = 'Can not deactivate user
that has valid book';

    END IF;

END

CREATE DEFINER=`root`@`localhost` TRIGGER
`hotel_system`.`rooms_BEFORE_INSERT` BEFORE INSERT ON `rooms` FOR
EACH ROW

BEGIN

```

/ 8 limitation from file*/*

IF new.price_per_night <= 40 THEN

SET new.room_type = 'BUDGET';

END IF;

IF new.price_per_night > 40 AND new.price_per_night <= 200 THEN

SET new.room_type = 'COMFORT';

END IF;

IF new.price_per_night > 200 AND new.price_per_night <= 500 THEN

SET new.room_type = 'BUSINESS';

END IF;

IF new.price_per_night > 500 THEN

SET new.room_type = 'LUXURY';

END IF;

END

CREATE DEFINER='root'@'localhost' TRIGGER

*`hotel\_system`.`rooms\_BEFORE\_UPDATE` BEFORE UPDATE ON `rooms` FOR
EACH ROW*

BEGIN

/ 8 limitation from file*/*

IF new.price_per_night <= 40 THEN

SET new.room_type = 'BUDGET';

END IF;

IF new.price_per_night > 40 AND new.price_per_night <= 200 THEN

SET new.room_type = 'COMFORT';

END IF;

IF new.price_per_night > 200 AND new.price_per_night <= 500 THEN

SET new.room_type = 'BUSINESS';

END IF;

IF new.price_per_night > 500 THEN

SET new.room_type = 'LUXURY';

END IF;

END

CREATE DEFINER= `root`@`localhost` TRIGGER

*`hotel\_system`.`rooms\_BEFORE\_DELETE` BEFORE DELETE ON `rooms` FOR
EACH ROW*

BEGIN

/ can't delete booked room (1 limitation from file)*/*

IF old.is_booked = 1 Then

signal sqlstate '45000' set message_text = 'Can not delete booked room';

END IF;

END

Основний код серверної частини

```
package com.ua.hotel.repository;
```

```
import com.ua.hotel.entity.BookingEntity;
```

```
import org.springframework.data.jpa.repository.JpaRepository;
```

```
import org.springframework.data.jpa.repository.Modifying;
```

```
import org.springframework.data.jpa.repository.Query;
```

```
import org.springframework.stereotype.Repository;
```

```
import java.sql.Date;
```

```
import java.util.List;
```

```
@Repository
```

```
public interface BookingRepository extends JpaRepository<BookingEntity, Long> {
```

```
@Modifying
```

```
@Query(value = "INSERT INTO booking(ID, customer_id, room_number, arrival,  
checkout, comment, persons_number) \n" +
```

```
"VALUES(:id, :customerId, :roomNumber, :arrival, :checkout, :comment,
:personsNumber);", nativeQuery = true)
```

```
void saveBooking(Long id, Long customerId, Long roomNumber, Date arrival,
    Date checkout, Integer personsNumber, String comment);
```

@Modifying

```
@Query(value = "UPDATE booking SET customer_id = :customerId,
room_number = :roomNumber, " +
```

```
"arrival = :arrival, checkout = :checkout, comment = :comment,
persons_number = :personsNumber " +
```

```
"WHERE ID = :id", nativeQuery = true)
```

```
void updateBooking(Long id, Long customerId, Long roomNumber, Date arrival,
    Date checkout, Integer personsNumber, String comment);
```

```
@Query(value = "SELECT * FROM booking WHERE DATEDIFF(arrival,
NOW()) > 0;", nativeQuery = true)
```

```
List<BookingEntity> selectAllBookingsWhereArrivalLaterThanNow();
```

```
@Query(value = "SELECT ID, booking.customer_id, room_number,
nights_quantity, arrival, " +
```

```
"checkout, comment, time_created, persons_number FROM booking " +
```

```
"INNER JOIN payments " +
```

```
"ON booking.ID = payments.booking_id " +
```

```
"WHERE payments.cancelled = :isCancelled", nativeQuery = true)
```

```
List<BookingEntity> getBookingsByCancellation(boolean isCancelled);
```

```
@Query(value = "SELECT * FROM booking\n" +
    "INNER JOIN payments ON payments.booking_id = booking.ID\n" +
    "WHERE DATEDIFF(NOW(), booking.arrival) > 0 AND\n" +
    "DATEDIFF(booking.checkout, NOW()) > 0 AND cancelled = 0",
    nativeQuery = true)
```

```
List<BookingEntity> getActiveCurrentBookings();
```

```
@Query(value = "SELECT * FROM booking WHERE customer_id =\n" +
    ":customerId", nativeQuery = true)
```

```
List<BookingEntity> getAllCustomerBookings(Long customerId);
```

```
}
```

```
package com.ua.hotel.repository;
```

```
import com.ua.hotel.entity.CancellationEntity;
```

```
import org.springframework.data.jpa.repository.JpaRepository;
```

```
import org.springframework.data.jpa.repository.Modifying;
```

```
import org.springframework.data.jpa.repository.Query;
```

```
import org.springframework.stereotype.Repository;
```

```
@Repository
```

```
public interface CancellationsRepository extends JpaRepository<CancellationEntity,
Long> {
```

```
    @Modifying
```

```
    @Query(value = "INSERT INTO cancellations(booking_id) \n" +
        "VALUES(:id);", nativeQuery = true)
```

```
    void cancelBooking(Long id);
```

```
}
```

```
package com.ua.hotel.repository;
```

```
import com.ua.hotel.entity.CustomerEntity;
```

```
import org.springframework.data.jpa.repository.JpaRepository;
```

```
import org.springframework.data.jpa.repository.Modifying;
```

```
import org.springframework.data.jpa.repository.Query;
```

```
import org.springframework.data.repository.query.Param;
```

```
import org.springframework.stereotype.Repository;
```

```
import java.sql.Date;
```

```
import java.sql.Timestamp;
```

```
import java.util.List;
```

```
@Repository
```

```
public interface CustomerRepository extends JpaRepository<CustomerEntity, Long>
{
```

```
    @Modifying
```

```
    @Query(value = "INSERT INTO customers(first_name, last_name, address,
post_code, city, country, email, phone) "
```

```
        +
```

```
"VALUES(:firstName,:lastName,:address,:postCode,:city,:country,:email,:phone);",
```

```
        nativeQuery = true)
```

```
    void saveCustomer(String firstName, String lastName, String address, String
postCode,
```

```
                        String city, String country, String email, String phone);
```

```
    @Modifying
```

```
    @Query(value = "UPDATE customers SET first_name = :firstName, last_name =
:lastName, address = :address," +
```

```
        "post_code = :postCode, city = :city, country = :country, email = :email,
phone = :phone, is_active = :active" +
```

```
        " WHERE ID = :id",
```

```
        nativeQuery = true)
```

```
    void updateCustomer(Long id, String firstName, String lastName, String address,
String postCode,
```

```
                        String city, String country, String email, String phone, boolean active);
```

@Modifying

*@Query(value = "UPDATE customers SET is_active = 0 WHERE ID = :id",
nativeQuery = true)*

void deactivateUser(@Param("id") Long id);

*@Query(value = "SELECT * FROM customers " +
"INNER JOIN booking ON booking.customer_id = customers.ID " +
"INNER JOIN rooms ON booking.room_number = rooms.ID " +
"INNER JOIN payments on booking.ID = payments.booking_id " +
"WHERE room_type = :roomType AND payments.cancelled = :isCancelled",
nativeQuery = true)*

*List<CustomerEntity> getCustomersByRoomTypeAndCancelled(String roomType,
boolean isCancelled);*

*@Query(value = "SELECT * FROM customers\n" +
"INNER JOIN booking ON booking.customer_id = customers.ID\n" +
"INNER JOIN payments ON payments.booking_id = booking.ID\n" +
"WHERE booking.checkout = :date AND payments.cancelled =
:isCancelled", nativeQuery = true)*

*List<CustomerEntity> getCustomerByCheckoutDateAndCancelled(Date date,
boolean isCancelled);*

*@Query(value = "SELECT * FROM customers\n" +
"INNER JOIN booking ON booking.customer_id = customers.ID\n" +*

"INNER JOIN payments ON payments.booking_id = booking.ID\n" +

*"WHERE booking.arrival = :date AND payments.cancelled = :isCancelled",
nativeQuery = true)*

*List<CustomerEntity> getCustomerByArrivalDateAndCancelled(Date date,
boolean isCancelled);*

*@Query(value = "SELECT DISTINCT(customers.ID), first_name, last_name,
email, post_code, address, city, country, phone, reg_time, customers.is_active FROM
customers\n" +*

"INNER JOIN booking ON booking.customer_id = customers.ID\n" +

"INNER JOIN rooms ON rooms.ID = booking.room_number\n" +

"INNER JOIN payments ON payments.booking_id = booking.ID\n" +

*"WHERE ((DATEDIFF(booking.arrival, :startDate) >= 0) AND
(DATEDIFF(:endDate, booking.checkout) >= 0)) \n" +*

*"AND rooms.room_type = :roomType AND payments.cancelled = 0",
nativeQuery = true)*

*List<CustomerEntity> getCustomersWithValidBookingsInPeriod(Date startDate,
Date endDate, String roomType);*

*@Query(value = "SELECT * FROM customers\n" +*

*"INNER JOIN cancellations ON cancellations.customer_id =
customers.ID\n" +*

*"WHERE timestampdiff(SECOND, :startDate, cancellations.cancel_time) >=
0 \n" +*

```
"AND timestampdiff(SECOND, cancellations.cancel_time, :endDate) >= 0",
nativeQuery = true)
```

```
List<CustomerEntity> getCustomersThatCancelledInPeriod(Timestamp startDate,
Timestamp endDate);
```

```
@Query(value = "SELECT * FROM customers WHERE email = :email",
nativeQuery = true)
```

```
CustomerEntity findByEmail(String email);
```

```
}
```

```
package com.ua.hotel.repository;
```

```
import com.ua.hotel.entity.RoomEntity;
```

```
import org.springframework.data.jpa.repository.JpaRepository;
```

```
import org.springframework.data.jpa.repository.Modifying;
```

```
import org.springframework.data.jpa.repository.Query;
```

```
import org.springframework.data.repository.query.Param;
```

```
import org.springframework.stereotype.Repository;
```

```
import java.sql.Date;
```

```
import java.util.List;
```

```
@Repository
```

```
public interface RoomsRepository extends JpaRepository<RoomEntity, Long> {
```

@Modifying

```
@Query(value = "INSERT INTO rooms(ID, price_per_night, max_persons)
VALUES(:id,:pricePerNight, :maxPersons);",
      nativeQuery = true)
```

```
void saveRoom(Long id, Double pricePerNight, Integer maxPersons);
```

@Modifying

```
@Query(value = "UPDATE rooms SET price_per_night = :pricePerNight, " +
      "max_persons = :maxPersons, is_active = :isActive WHERE ID = :id",
      nativeQuery = true)
```

```
void updateRoom(@Param("id") Long id, Double pricePerNight, Integer
maxPersons, boolean isActive);
```

```
@Query(value = "SELECT * FROM rooms WHERE is_booked = 0;", nativeQuery
= true)
```

```
List<RoomEntity> getAllNotBookedRooms();
```

```
@Query(value = "SELECT Distinct(rooms.ID), room_type, price_per_night,
max_persons, is_booked, is_active FROM rooms\n" +
```

```
"LEFT JOIN booking ON booking.room_number = rooms.ID\n" +
```

```
"LEFT JOIN payments ON payments.booking_id = booking.ID\n" +
```

```
"WHERE (rooms.is_booked = 0 AND DATEDIFF(:startDate, NOW()) >= 0
AND DATEDIFF(:endDate, :startDate) >= 0)\n" +
```

```
"OR ((DATEDIFF(booking.arrival, :endDate) > 0 OR
DATEDIFF(:startDate, booking.checkout) > 0) \n" +
```

```
"AND DATEDIFF(:startDate, NOW()) >= 0 AND DATEDIFF(:endDate,
:startDate) >= 0 AND payments.cancelled = 0)",
```

```
nativeQuery = true)
```

```
List<RoomEntity> getFreeRoomsInPeriod(Date startDate, Date endDate);
```

```
@Query(value = "SELECT avg(price_per_night) FROM rooms WHERE
room_type = :roomType", nativeQuery = true)
```

```
double getAvgRoomPriceBuType(String roomType);
```

```
}
```

```
package com.ua.hotel.repository;
```

```
import com.ua.hotel.entity.PaymentEntity;
```

```
import org.springframework.data.jpa.repository.JpaRepository;
```

```
import org.springframework.stereotype.Repository;
```

```
@Repository
```

```
public interface PaymentRepository extends JpaRepository<PaymentEntity, Long> {
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<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>org.example</groupId>
    <artifactId>hotel_system</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.8.RELEASE</version>
    </parent>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>

```

```
<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-devtools</artifactId>

</dependency>

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

    <groupId>org.projectlombok</groupId>

    <artifactId>lombok</artifactId>

    <version>1.16.8</version>

    <scope>provided</scope>

</dependency>

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>
```

```
<dependency>
```

```
  <groupId>mysql</groupId>
```

```
  <artifactId>mysql-connector-java</artifactId>
```

```
  <version>8.0.16</version>
```

```
  <scope>runtime</scope>
```

```
</dependency>
```

```
<dependency>
```

```
  <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-jdbc</artifactId>
```

```
</dependency>
```

```
</dependencies>
```

```
</project>
```

ДОДАТОК В

Список задач, що вирішує програмний продукт

1. Додати користувача
2. Додати кімнату
3. Створити бронь
4. Скасувати бронь
5. Вибрати всі броні, час прибуття яких пізніше поточної дати
6. Вибрати всі вільні кімнати
7. Вибрати всіх користувачів у яких є бронь (скасована / не скасована) з певним типом кімнат
8. Вибрати всі скасовані / не скасовані броні
9. Вибрати всіх користувачів з такою-то датою прибуття / відправлення (активні / неактивні броні)
10. Показати всі вільні номери на такі-то дати
11. Вибрати все броні користувача
12. Вивести дані всіх користувачів, які забронювали номери певного типу в такий-то період (не скасована бронь)
13. Вивести дані всіх користувачів, які скасували бронь номерів в такий-то період
14. Вивести середню ціну за ніч для номерів певного типу
15. Вибрати всі активні броні, які зараз тривають

16. Знайти користувача по email
17. Вибрати всіх користувачів
18. Вибрати всі кімнати
19. Знайти користувача по ID
20. Редагувати користувача
21. Знайти кімнату по ВД
22. Редагувати кімнату
23. Вибрати все броні
24. Редагувати бронь
25. Знайти бронь по ID
26. Вивести усі оплати
27. Вивести усі відміни

Список основних тригерів, які реалізовані

1. Не можна видаляти кімнату, якщо в ній є користувач
2. Початок проживання в номері не може бути раніше ніж час проведення оплати (при створенні броні створюється оплата)
3. В кімнату не можна заселити більше доступного числа людей
4. Не можна деактивувати користувача, якщо на даний момент не закінчився термін перебування нескасованого бронювання
5. У разі створення броні кімната стає заблокованою, а користувач, що забронював, не може бронювати на ці дати

6. Дати нескасованих броні одного і того ж номера не повинні перетинатися
7. Дати нескасованих бронювань одного і того ж користувача не повинні перетинатися
8. Для кожної категорії номерів не повинен порушуватися певний ціновий діапазон
9. У разі скасування броні відповідні зміни повинні відбуватися: відображення в Оплаті, кімната стає вільна, користувач може забронювати на ці дати