Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

Освітній ступінь – бакалавр

на тему: **«РОЗРОБКА СИСТЕМИ ДЛЯ АВТОМАТИЧНОЇ НОРМАЛІЗАЦІЇ СТРУКТУРИ БАЗИ ДАНИХ»**

Виконав: студент 4-го року навчання,

Спеціальності

122 Комп'ютерні науки

Новак Володимир Ігорович

Керівник Яремко С. А.,_____

асистент

Рецензент _____

(прізвище та ініціали)

Кваліфікаційна робота захищена з оцінкою _____

Секретар ЕК _____

«___» _____ 20 ___ р.

Київ – 20___

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри інформатики,

проф., д.ф.-м.н.

_____М. М. Глибовець

(підпис)

„____” _____2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студенту Новаку В. І. факультету інформатики 4-го курсу

ТЕМА Розробка системи для автоматичної нормалізації структури бази даних

Вихідні дані:

- Веб-застосунок з реалізованим в ньому алгоритмом нормалізації

Зміст ТЧ до курсової роботи:

Індивідуальне завдання

Вступ

1 Теоретичні аспекти нормалізації

2 Аналіз поточного стану нормалізації баз даних

3 Процес розробки системи для автоматичної нормалізації структури бази даних

4 Тестування розробленої системи нормалізації

Висновки

Список використаної літератури та електронних ресурсів

Дата видачі „___” _____ 2023 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Тема: Розробка системи для автоматичної нормалізації структури бази даних

Календарний план виконання роботи:

№ п/п	Назва етапу виконання кваліфікаційної роботи	Термін виконання етапу	Примітка
1.	Отримання завдання.	17.10.2023	
2.	Ознайомлення з теоретичним матеріалом в технічних джерелах.	15.02.2024	
3.	Пошук та аналіз існуючих інструментів автоматичної нормалізації.	03.03.2024	
4.	Написання текстової частини про теоретичні засади нормалізації та можливі інструменти.	15.03.2024	
5.	Розробка власного застосунку для автоматичної нормалізації.	25.04.2024	
6.	Попередній захист роботи.	15.05.2024	
7.	Написання текстової частини про розробку застосунку для нормалізації та тестування.	16.05.2024	
8.	Створення презентації та написання доповіді.	21.05.2024	
9.	Захист проекту.	31.05.2024	

Зміст

Анотація 5

ВСТУП..... 6

РОЗДІЛ 1: ТЕОРЕТИЧНІ АСПЕКТИ АВТОМАТИЧНОЇ НОРМАЛІЗАЦІЇ..... 9

1.1 Основні принципи нормалізації баз даних 9

1.2 Алгоритми нормалізації баз даних 11

1.3 Методи оптимізації процесу нормалізації 14

РОЗДІЛ 2: АНАЛІЗ ПОТОЧНОГО СТАНУ НОРМАЛІЗАЦІЇ БАЗ ДАНИХ 17

2.1 Огляд існуючих методів нормалізації 17

2.2 Проблеми нормалізації в сучасних системах управління базами даних 20

РОЗДІЛ 3: ПРОЦЕС РОЗРОБКИ СИСТЕМИ ДЛЯ АВТОМАТИЧНОЇ НОРМАЛІЗАЦІЇ СТРУКТУРИ БАЗИ ДАНИХ 22

3.1 Вибір технологій та інструментів 22

3.2 Реалізація ключових операцій алгоритму 23

3.3 Реалізація ключових операцій зі сторони клієнта 33

РОЗДІЛ 4: ТЕСТУВАННЯ РОЗРОБЛЕНОЇ СИСТЕМИ НОРМАЛІЗАЦІЇ 39

4.1 Автоматизоване тестування програми 39

4.2 Тестування роботи алгоритму 40

ВИСНОВКИ 44

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ ТА ЕЛЕКТРОННИХ РЕСУРСІВ . 46

Анотація

Дана робота розкриває основні теоретичні засади теми нормалізації, включаючи основні поняття та можливі алгоритми декомпозиції до різних нормальних форм. Також було проведено пошук вже існуючих онлайн-інструментів для нормалізації структур баз даних, та аналіз можливостей нормалізації в системах управління базами даних.

У наступних розділах було імплементовано логіку алгоритму для нормалізації структури бази даних, а саме декомпозицію поданої структури в вигляді множини функціональних залежностей до реляцій третьої нормальної форми, а також і різні допоміжні операції, включаючи знаходження мінімального покриття поданих функціональних залежностей, перевірки на властивості з'єднання без втрат та збереження залежностей. Цю імплементацію було подано у вигляді веб-застосунку, розробленого з використанням фреймворків React.js та Node.js.

Для розробки алгоритму було використано основні поняття нормалізації з підручників для вивчення курсу баз даних та матеріал, засвоєний на заняттях курсу «Бази даних та інформаційні системи», пройденого на 2-му році навчання.

ВСТУП

З розвитком цифрових технологій, які почали проникати в усі можливі аспекти сучасного життя, необхідність зберігати, обробляти та діставати з серверів великі об’єми даних все більше зростає. Незважаючи на розвиток та популяризацію нереляційних баз даних, реляційні так і залишаються більш поширеними для використання в різних програмах та застосунках завдяки перевіреності часом та можливості надсилання на сервер структурованих запитів. Постійний розвиток реляційних баз даних є надзвичайно важливим не лише для забезпечення цілісності та ефективності даних, а й для підвищення продуктивності їхньої роботи. Для досягнення саме цієї мети існує такий основний метод, як нормалізація баз даних – процес, що спрямований на мінімізацію надмірності різних залежностей та загалом оптимізацію проектування бази даних.

Вибір даної теми для поточної дипломної роботи можна обґрунтувати власним спостереженням наявності малої кількості застосунків чи інструментів для автоматизованого процесу нормалізації певної структури бази даних, до того ж, розробники при побудові архітектури баз даних часто можуть нехтувати цим процесом, що знову ж таки зумовлено відсутністю інструментів для нормалізації баз даних. Невикористання алгоритму нормалізації може бути причиною створення більш неефективної системи бази даних, які в подальшому будуть менш надійними, особливо якщо такі бази даних міститимуть доволі великі об’єми даних. Саме тому якісна автоматизація процесу нормалізації структури баз даних може бути доволі корисною.

Актуальність даної теми можна пояснити постійною необхідністю використовувати бази даних у різних масштабних проектах, і важливо, щоби їх робота була оптимізована, і їх нормалізація буде для цього в нагоді. Наукове значення теми полягає в детальному дослідженні процесу нормалізації структури баз даних, в той час, як практичне значення теми – в створенні програмного

застосунку, що виконуватиме той процес, для якого не розроблено велику кількість програмного забезпечення.

Об’єктом дослідження даної роботи є автоматизація процесу нормалізації структури баз даних, чий вибір можна пояснити знову ж таки важливістю та корисністю процесу нормалізації при побудові архітектури великих проектів, а також відсутністю великої кількості застосунків, які би виконували цей процес автоматично. Предмет дослідження даної роботи – це алгоритм, що використовується для автоматизації процесу нормалізації.

Основною метою дослідження є розробка такої системи, що буде автоматично проводити нормалізацію структури поданої бази даних. Серед завдань дослідження потрібно зазначити такі:

- Аналіз поточних інструментів для нормалізації бази даних
- Розробка програмного алгоритму для нормалізації поданої бази даних
- Реалізація розробленого алгоритму у вигляді повноцінного застосунку

Основним джерелом для праці над поточною роботою є матеріал, здобутий на нормативному предметі «Бази даних та інформаційні системи», що відбувся на 2 курсі, а також серед списку літератури є різні навчальні підручники, розроблені для проходження курсу по роботі з базами даних.

В методологічну основу даної роботи входить теоретичний аналіз існуючих методів нормалізації баз даних та загалом цього процесу.

Дана робота складається з 4 розділів.

Перший розділ розглядає такі теоретичні аспекти процесу нормалізації, як нормальні форми та алгоритми переведення реляції до третьої нормальної форми, вивчені протягом навчального курсу про бази даних та досліджені в Інтернеті.

Другий розділ присвячений огляду різних онлайн-інструментів для нормалізації структури баз даних та наявності можливості автоматизованої нормалізації у різних системах управління базами даних.

Третій розділ детально описує процес розробки власного застосунку для нормалізації структури баз даних, включаючи створені алгоритми, використані інструменти для цього та його кінцевий вигляд.

Четвертий розділ присвячений аналізу роботи та результатів створеного застосунку, в особливості дослідження перевірок нормальних форм на властивості з’єднання без втрат та збереження залежностей, вивчені протягом навчального курсу про бази даних.

РОЗДІЛ 1: ТЕОРЕТИЧНІ АСПЕКТИ АВТОМАТИЧНОЇ НОРМАЛІЗАЦІЇ

1.1 Основні принципи нормалізації баз даних

Нормалізація бази даних - це систематизований підхід до побудови архітектури бази даних, який використовується для ефективної організації її структури. Впорядковуючи дані в певні «логічні групи», кожна з яких описує невелику частину всього набору даних, нормалізація передбачає мінімізацію надмірності даних і забезпечення цілісності даних в майбутньому. Ця практика глибоко вкорінена в теорії функціональних залежностей, і використовує концепції нормальних форм, які в свою чергу мають вирішальне значення для визначення способу доступу до інформації та її збереження в системі баз даних.

Говорячи про функціональні залежності, то вони є надзвичайно важливими для розуміння структурних зв'язків у базі даних. Функціональна залежність, виражена як X → Y, вказує на те, що якщо два рядки даних збігаються за атрибутами з множини X, то вони також повинні узгоджуватися за атрибутами у множині Y. Концепція функціональних залежностей має вирішальне значення для підтримання узгодженості даних у базі даних. Наприклад, візьмемо таблицю зі стовпцями EmployeeID, EmployeeName та Department, де є функціональна залежність EmployeeID → EmployeeName, яка безпосередньо вказує на те, що кожному EmployeeID (ідентифікатору) відповідає рівно одне значення зі стовпця EmployeeName (ім’я та прізвище одного працівника). Така функціональна залежність забезпечує узгодженість даних, уникаючи ситуацій, в яких одне і те саме значення EmployeeID може відповідати декільком значенням EmployeeNames.

Також при побудові певної нормальної форми часто вживається такий термін, як реляція (відношення). Терміном «реляція» називають по факту таблицю, що складається з рядків і стовпців, де кожен рядок (кортеж) являє собою один запис, а кожен стовпець – один атрибут. Структура реляцій

визначається їхньою схемою, яка в свою чергу керується “правилами”, що випливають з наявних функціональних залежностей. Також реляції є основою для ефективного зберігання даних і оптимізації пошуку цих ж даних у відповідь на запити користувачів. Правильно структурована реляція відповідає принципам, які гарантують, що всі її значення є атомарними, кожен атрибут є унікальним, а порядок рядків і стовпців не має значення для роботи бази даних. [1]

Повертаючись до опису такого вищезгаданого терміну, як нормальні форми, можна зазначити, що якщо певна реляція виконує певні необхідні умови, то вона знаходиться у n-тій нормальній формі, з чого випливає, що кожен тип нормальної форми має свої умови для знаходження реляції у ній. Розглянемо стисло існуючі рівні нормалізації (нормальні форми):

- 1-ша нормальна форма: реляція знаходиться у ній, якщо кожна комірка таблиці містить лише одне значення, тобто є атомарною, а кожен запис - унікальний.
- 2-га нормальна форма: реляція вже знаходиться у першій нормальній формі, і також всі її вторинні (неключові) атрибути повністю залежать від первинного ключа.
- 3-тя нормальна форма: реляція вже знаходиться у другій нормальній формі, і також не містить транзитивних функціональних залежностей вторинних атрибутів від первинного ключа.
- Нормальна форма Бойса-Кодда (названа в честь двох інформатиків, що дослідили певні аномалії в 3-тій нормальній формі): реляція вже знаходиться в третій нормальній формі, і також у кожній функціональній залежності детермінант (ліва частина залежності, від якої залежить права) є потенційним ключем.

До того ж, існують і подальші нормальні форми, проте вони досить рідко використовуються, а саме 4-та нормальна форма та 5-та нормальна форма, що випливають зі знайдених аномалій попередніх форм, але у даній роботі вони не будуть детально розглянуті. [2]

В особливих випадках, якщо нормалізація спрямована на забезпечення цілісності даних, то денормалізація передбачає додавання навмисної надмірності до бази даних для підвищення продуктивності запитів, особливо в системах, де переважають операції читання, однак, цей процес може легко зіпсувати роботу раніше нормалізованої реляції.

Застосування нормалізації передбачає ретельний розгляд впливу кожної її форми на цілісність даних та продуктивність запитів. Вибір певної нормальної форми повинен відображати конкретні вимоги, гарантуючи, що структура бази даних ефективно підтримує як поточні, так і підтримуватиме майбутні потреби.

1.2 Алгоритми нормалізації баз даних

При виборі нормальної форми для нормалізації структури бази даних немає чітких критеріїв, тому потрібно розглядати кожен варіант індивідуально. Хоч і для кожної нормальної форми є свій алгоритм переведення в неї, проте насправді вони практично однакові. Ці алгоритми не тільки підтримують теоретичні основи, встановлені в нормальних формах, але й забезпечують практичну застосовність в реальних системах баз даних.

Отже, першим кроком у будь-якому алгоритмі нормалізації є встановлення всіх функціональних залежностей у базі даних. Часто для цього потрібно окремо переглядувати взаємозв'язки кожного атрибуту з іншими, щоб вивести потенційні залежності. До того ж, в залежності від предметної області бази даних, може бути такий варіант, що не всі залежності можна виявити виключно за допомогою аналізу, особливо у великих або складних наборах даних. Для точного визначення всіх релевантних залежностей дуже важлива допомога розробників баз даних або експертів з предметної області. Саме тому зазвичай в різних алгоритмах нормалізації на вхід подається не просто ER-модель бази даних, а вже її вирахувана множина функціональних залежностей.

Після визначення вхідної множини функціональних залежностей наступним кроком є обчислення мінімального покриття. Мінімальне покриття

набору функціональних залежностей - це набір, еквівалентний вхідному, але з якомога меншою кількістю залежностей, кожна з яких має мінімальну кількість атрибутів. Знаходження та подальше використання мінімального покриття допомагає забезпечити ефективність та результативність процесу нормалізації за рахунок спрощення залежностей, причому без втрати інформації.

Обчислення мінімального покриття зазвичай включає в себе три основні кроки:

1. Знаходження одноатрибутної правої частини

Даний крок є найпростішим, оскільки кожна функціональна залежність типу $X \rightarrow Y_1Y_2\ldots Y_n$ (де $n > 1$) видаляється з множини та замість неї додаються нові n залежностей, в яких детермінантом буде X, а залежними аргументами – Y_i ($i <= n$) з видаленої залежності.

2. Антинадлишковість функціональних залежностей

На даному кроці береться кожна залежність $X \rightarrow Y$ з множини F, і видаляється з неї (нехай множина F без залежності $X \rightarrow Y$ буде F_i). Тоді потрібно зробити перевірку умови, що $F \sim F_i$, тобто, чи ці дві множини є еквівалентними. Вочевидь, F буде накриттям F_i, а чи працює це в іншу сторону? Для цього знаходимо покриття X на множину F, і якщо Y належатиме цьому покриттю, тоді множину $X \rightarrow Y$ видаляють з множини F.

3. Антинадлишковість лівої частини залежностей

Для цього також береться кожна залежність $X \rightarrow Y$ з множини F. Нехай замість одного атрибуту X в лівій частині буде n атрибутів ($X_1X_2\ldots X_n \rightarrow Y$), для кожного атрибута лівої частини тимчасово видаляється X_i і обчислюється замикання нової лівої частини (позначимо X’). Тут ми так само перевіряємо еквівалентність множин F та F з X` замість X. Якщо Y все ще знаходиться у замиканні X' відносно множини F та у замиканні X відносно еквівалентної йому, то X_i є зайвим і його можна назавжди вилучити з лівої частини.

До моменту, поки не залишиться залежностей, які можна буде видалити, не руйнуючи зберігання залежностей у множині F, кроки 2 і 3 можна повторювати.

Далі, декомпозиція схеми бази даних до певної нормальної форми є надважливим етапом нормалізації бази даних, який забезпечує цілісність даних, мінімізує надмірність і підвищує її продуктивність. Цей процес передбачає застосування повторення низки алгоритмічних перетворень для перетворення схеми бази даних у реляції, які задовольняють певним умовам, відповідним обраній нормальній формі.

Загалом, для декомпозиції отриманого мінімального покриття набору функціональних залежностей до 2 нормальної форми, 3 нормальної форми чи нормальної форми Бойса-Кодда використовується такий алгоритм:

1. Реляція перевіряється на відповідність обраній нормальній формі, якщо вона їй не відповідає, то використовується теорема Хіта для декомпозиції однієї з залежностей в окрему реляцію. Теорему Хіта можна описати таким чином: якщо в реляції R з наборами атрибутів задано таку функціональну залежність $A \rightarrow B$, що $B \nsubseteq A$ та квазіключ реляції не належить атрибуту (або їх набору) A, то можна декомпонувати реляцію на дві підреляції – R (A, B) та реляція з іншими атрибутами, де виключено B з атрибутів та залежності, які містять B.
2. Відокремлена залежність стає окремою реляцією, а в попередній видаляється права частина видаленої залежності зі списку атрибутів та видаляються залежності, які містили цю праву частину. Важливо відмітити, що для 3 нормальної форми та нормальної форми Бойса-Кодда, якщо в попередній є неявна транзитивна залежність від ключа, то кроки 1 і 2 відміняються, і алгоритм починається з початку, але вже береться інша залежність, яка підходить для декомпозиції.
3. Кроки 1 і 2 повторюються, поки реляція, з якої ми декомпонували інші залежності, сама не буде відповідати обраній нормальній формі.

Після застосування вищезгаданих алгоритмів нормалізації важливо перевірити отриману схему реляцій, щоб переконатися, що вона відповідає всім вимогам відповідних нормальних форм, для цього потрібно використати перевірку виконання властивостей з’єднання без втрат та збереження залежностей. Якщо було обрано 3 нормальну форму для зведення, то властивість збереження залежностей може не завжди виконуватись, а от з’єднання без втрат повинно завжди виконуватись для отриманих реляцій.

Для перевірки властивості з’єднання без втрат можна використати метод таблиці таким чином: будується таблиця, рядки якої відповідають отриманим у результаті поділу реляціям, а стовпці – аргументам початкової реляції A_1,..., A_n. Комірка заповнюється символами a_j, якщо елемент, розташований в і-му рядку і j-му стовпці, відповідає атрибуту A_i реляції R_j, інакше в комірці записується B_{ij}. Після заповнення таблиці здійснюється перегляд усіх атрибутів, що розташовані в іншій частині кожної функціональної залежності X → Y. Якщо для атрибутів в X знаходяться рядки, де у відповідних місцях розташовані a_j, то елементи b_{ik} цих рядків, відповідні символам атрибутів у Y, доповнюються символами a_k. Якщо в результаті з'явиться такий рядок, повністю заповнений a_j, то з'єднання виконано без втрат, інакше – ні.

Щоби виконати перевірку властивості збереження залежностей, потрібно визначити проекції відповідних залежностей на кожну отриману реляцію, а потім перевірити, чи об'єднання цих спроектованих залежностей у всіх декомпонованих реляціях все ще може забезпечити виконання кожної вихідної залежності. Для цього треба обчислити замикання наборів атрибутів X з кожної функціональної залежності X → Y за допомогою об’єднань залежностей декомпонованих реляцій. Якщо Y входить до замикання X, то залежність зберігається, інакше – ні. [3]

1.3 Методи оптимізації процесу нормалізації

Для алгоритмів нормалізації баз даних алгоритмічні вдосконалення можуть відіграти вирішальну роль у підвищенні ефективності та результативності процесів трансформації схем. Ці вдосконалення мають бути призначені для спрощення аналізу залежностей, мінімізації надмірності чи оптимізації стратегій декомпозиції. Було проведено невелике пошукове дослідження, за результатами якого було знайдено досить малу кількість розв'язків задачі спрощення процесу нормалізації баз даних, тому варто розглянути знайдені способи.

Доволі нещодавно було винайдено та опубліковано в Інтернеті підхід знаходження функціональних залежностей на основі решітки, який поступово досліджує усі комбінації атрибутів для виявлення функціональних залежностей. Цей метод структурує набори атрибутів у вигляді решітки, що дозволяє структуровано та поетапно досліджувати можливі залежності.

Кроки даного алгоритму:

1. Почати з найменших комбінацій атрибутів та ініціалізувати структуру решітки.
2. Для кожного вузла решітки (кожної комбінації атрибутів) перевірити, чи є він функціональною залежністю.
3. Якщо залежність знайдено, обрізати решітку, щоб видалити всі підмножини підтвердженої залежності, оскільки вони тепер не можуть бути мінімальними.
4. Перейти на наступний рівень решітки і повторити процес, поки всі комбінації атрибутів не будуть перевірені або відсічені.

Такий підхід зменшує обчислювальну надмірність, зосереджуючись лише на тих комбінаціях атрибутів, які з найбільшою ймовірністю утворюють залежності. Він відкидає усі можливі зайві обчислення, що робить його придатним для оброблення великих наборів даних. [4]

Отже, у даному розділі було детально розглянуто теоретичні основи та методології, необхідні для розуміння та ефективного впровадження нормалізації

баз даних, починаючи з головних термінів і базових алгоритмів. Також було проаналізовано різні моменти, що стосуються цієї теми, включаючи різні нормальні форми, способи перевірки правильності роботи алгоритму декомпозиції, та можливі способи оптимізації роботи з нормалізацією бази даних.

РОЗДІЛ 2: АНАЛІЗ ПОТОЧНОГО СТАНУ НОРМАЛІЗАЦІЇ БАЗ ДАНИХ

2.1 Огляд існуючих методів нормалізації

Як було зазначено раніше, нормалізація є досить важливим і корисним процесом в процесі управління базами даних для підвищення цілісності та ефективності даних. В Інтернеті існує всього декілька інструментів для нормалізації баз даних. Це можна пояснити кількома факторами, які впливають на використання цього алгоритму в управлінні базами даних.

Хоча нормалізація досить детально вивчається як фундаментальна концепція в курсі баз даних, її практичне застосування може значно відрізнятися в реальних сценаріях, де продуктивність і масштабованість часто мають пріоритет над суворим дотриманням нормальних форм. Це може пояснити те, чому розробники не приділяють достатньої уваги розробці таких інструментів, орієнтованих виключно на нормалізацію, оскільки практики віддають перевагу інструментам, що нехтують цим алгоритмом і пропонують більш розширений функціонал для оптимізації та загалом управління базами даних.

Крім того, небезпечно нормалізувати базу даних, коли вона вже існує і містить багато даних, оскільки є ймовірність втратити або пошкодити вже існуючі дані, тому краще нормалізувати її, коли сама структура бази даних знаходиться ще на стадії планування.

Потреби в нормалізації можуть суттєво відрізнятися залежно від конкретної бази даних, сценарію її використання та загалом потреб організації. Для багатьох підприємств повна нормалізація не завжди бажана, оскільки вона може призвести до надмірної фрагментації бази даних, що може значно погіршити продуктивність. Замість цього розробники баз даних часто віддають перевагу ручному втручанню та аналізу структури бази даних, щоб переконатися, що нормалізована структура відповідає як технічним, так і бізнес-вимогам. [5]

Отже, розглянемо існуючі онлайн-інструменти нормалізації, які було знайдено в Інтернеті:

1. "Functional Dependencies Checker".

Цей веб-додаток був написаний на мовах програмування Python та Javascritpt і розміщений на GitHub зареєстрованим користувачем з ніком "arjo129", тобто він має відкритий вихідний код. Вікно інструменту виглядає таким чином:

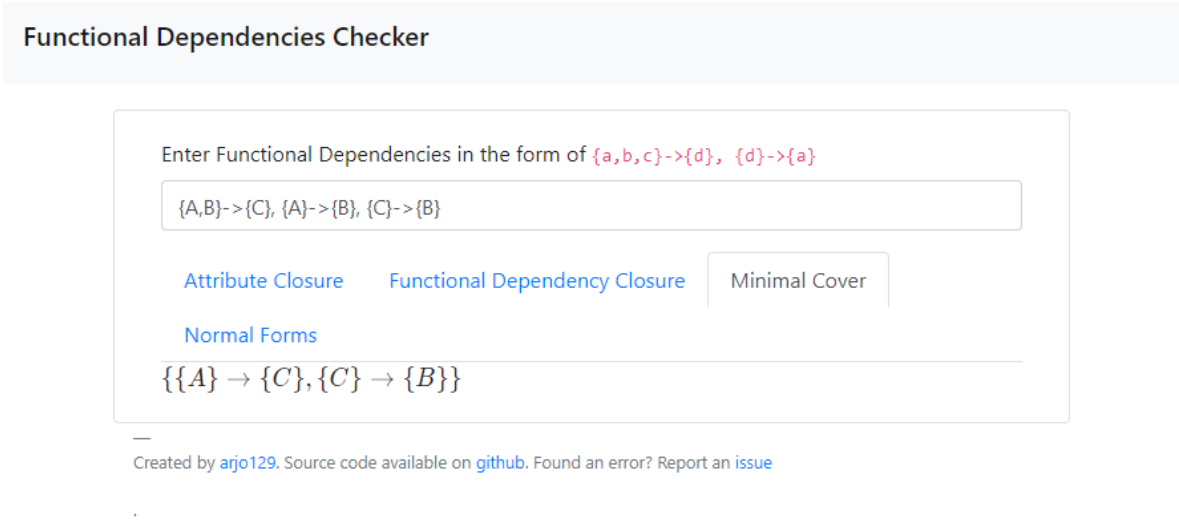


Рис. 2.1.1. Вікно застосунку "Functional Dependencies Checker" [6]

За допомогою даного застосунку можна обчислити замикання різних наборів атрибутів, і тим самим визначити потенційні ключі та суперключі, а також перевірити, які функціональні залежності входитимуть в замикання поданої множини, знайти мінімальне покриття та перевірити, чи подана множина буде належати певній нормальній формі (є перевірка на 2 нормальну норму, 3 нормальну форму та нормальну форму Бойса-Кодда). З недоліків можна виділити відсутність переведення поданої множини залежностей в певну нормальну форму. З іншого боку, даний застосунок працює доволі швидко, та має декілька корисних вищезгаданих функцій.

2. "Relational Database Tools"

Даний веб-сайт розроблено на мові програмування Java користувачем GitHub під нікнеймом “raymondcho”. Його клієнтська частина виглядає таким чином:

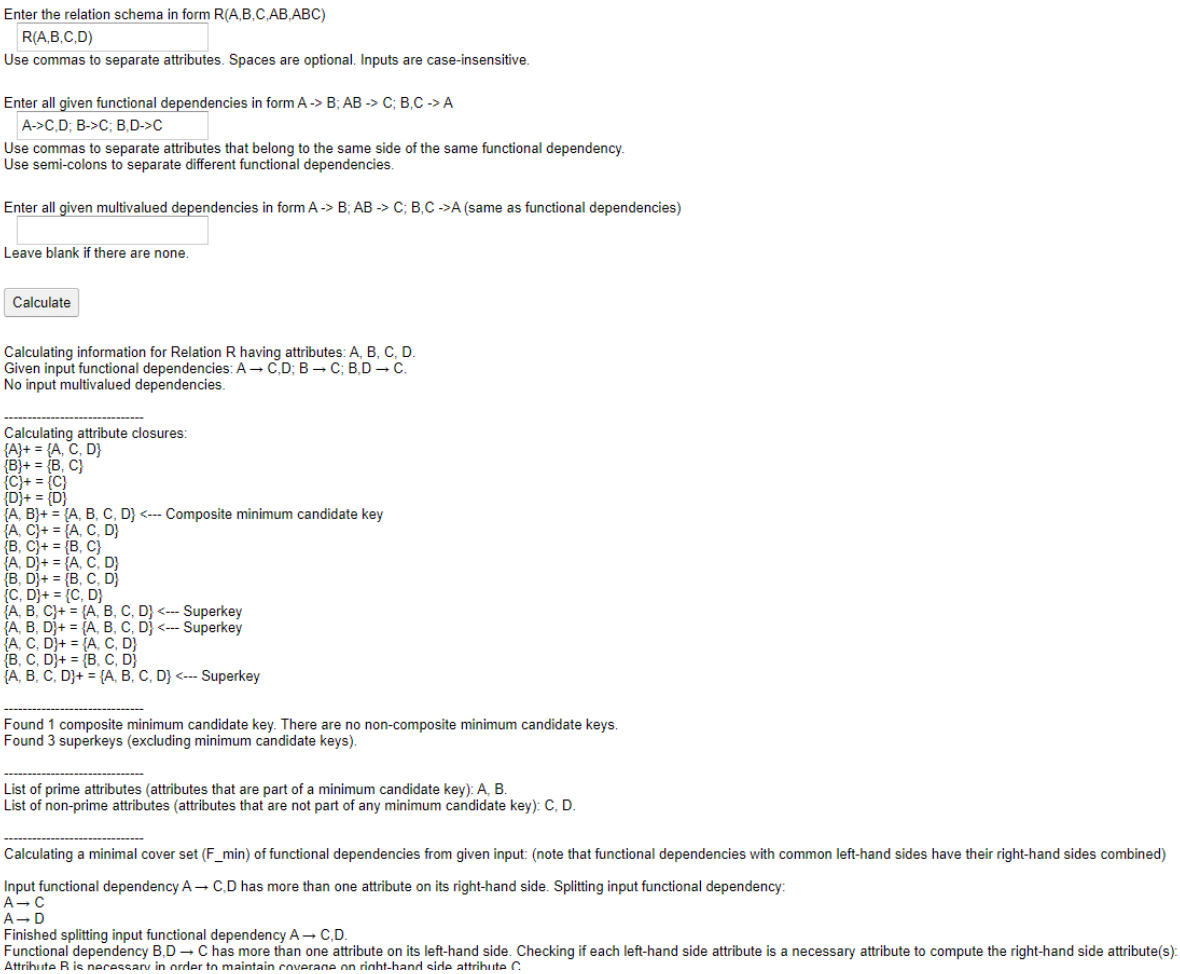


Рис. 2.1.2. Вікно застосунку “Relational Database Tools” [7]

Даний веб-сайт проводить усі базові операції для процесу нормалізації бази даних, включаючи знаходження мінімального покриття множини, знаходження суперключів та потенційних ключів, перевірка на відповідність деяким нормальним формам, та декомпозиція до нормальних форм. До різних операцій розписано процес знаходження результату, проте як недолік можна виділити відсутність певного інтуїтивного дизайну, адже на сайті є фактично декілька текстових полів та велика кількість недиференційованого тексту, до того ж, спочатку може бути не зовсім зрозуміла різниця, як вводити звичайні або багатозначні залежності в поля.

3. “Boyce-Codd relation solver”

Це спеціалізований інструмент для декомпозиції схем баз даних у нормальну форму Бойса-Кодда. Він розроблений JavaScript для обробки вхідних залежностей, а потім забезпечує покрокову декомпозицію в BCNF, показуючи кожен крок процесу. Даний застосунок також знаходиться на GitHub, розроблений користувачем «gamgi». Він має такий вигляд:

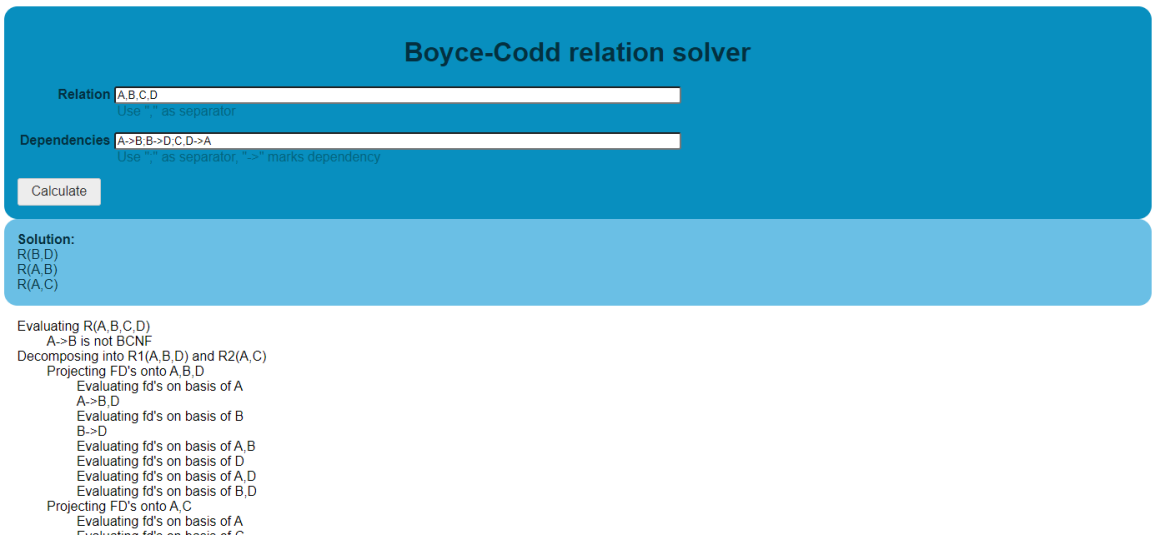


Рис. 2.1.3. Вікно застосунку “Boyce-Codd relation solver” [8]

Єдине, що можна зробити на цьому сайті – перевести множину функціональних залежностей в нормальну форму Бойса-Кодда. Відсутність інших можливих операцій обмежує функціонал сайту, що є певним недоліком, якщо розглядати цей сайт як інструмент для нормалізації структури бази даних.

2.2 Проблеми нормалізації в сучасних системах управління базами даних

Використання алгоритму нормалізації в сучасних СУБД, як уже згадувалося в попередньому підрозділі, пов'язане з низкою аномалій, які можуть впливати на продуктивність і зручність використання системи баз даних у реальних умовах.

Як відомо, алгоритми нормалізації часто розбивають одну реляцію на багато менших. Хоча це зменшує надмірність даних і покращує цілісність, велика кількість таблиць збільшує складність запитів. Потреба в декількох об'єднаннях для отримання пов'язаних даних може призвести до сповільнення часу відповіді на запит, особливо у великих базах даних. Ця проблема може стати досить гострою для транзакційних систем реального часу, для яких продуктивність і швидкість отримання даних є критично важливими.

Крім того, суворе застосування правил нормалізації може виявитися занадто негнучким для практичних потреб. Кожен рівень нормалізації спрямований на "нейтралізацію" певних видів аномалій, і чим складнішими є нормальні форми, тим складнішою стає структура бази даних. Пошук балансу між бажаним рівнем нормалізації та оптимальною продуктивністю для конкретних потреб додатків вимагає додаткового самоаналізу і зазвичай передбачає умовний компроміс, коли певна кількість операцій денормалізації допускається заради підвищення продуктивності. Мінливий характер бізнес-вимог часто вимагає постійного перегляду схеми бази даних, що робить нормалізовані бази даних складними в обслуговуванні та масштабуванні. Можна нормалізувати або денормалізувати базу даних, надсилаючи до неї користувацькі запити, які змінюють її структуру, але знову ж таки, потрібно розуміти, що якщо запит побудовано неправильно, структура бази даних може бути спотворена, або, якщо обсяг даних великий, такі запити будуть дуже повільними. [9]

Отже, хоча нормалізація є потужним інструментом оптимізації проектування баз даних, її застосування в сучасних СУБД не є безпроблемним. Основною проблемою, пов'язаною з проектуванням та експлуатацією сучасних систем баз даних, є компроміс між продуктивністю, складністю та потребами в обслуговуванні. Навряд чи можна зараз знайти опціонал для нормалізації бази даних в сучасних СУБД, але в Інтернеті є доступні онлайн-інструменти для нормалізації структури бази даних, причому доволі непогані.

РОЗДІЛ 3: ПРОЦЕС РОЗРОБКИ СИСТЕМИ ДЛЯ АВТОМАТИЧНОЇ НОРМАЛІЗАЦІЇ СТРУКТУРИ БАЗИ ДАНИХ

3.1 Вибір технологій та інструментів

Вибір відповідних технологій та інструментів є досить важливим при розробці такої програми, як система автоматичної нормалізації структури баз даних. Даний підрозділ присвячений обґрунтуванню вибору фреймворків React.js та Node.js як основних технологій для цього проекту.

React.js - потужний JavaScript фреймворк для створення користувацьких інтерфейсів, відомий своєю ефективністю та гнучкістю в роботі з динамічними даними, що змінюються. Для системи, яка робить можливим автоматизацію нормалізації баз даних, React.js може принести численні переваги. Однією з особливостей Document Object Model (DOM) у React є те, що користувацький інтерфейс може швидко оновлюватися у відповідь на зміни даних. Це надзвичайно важливо для процесів нормалізації, де різні значення динамічно включаються або виключаються, наприклад, додавання або видалення функціональних залежностей або оновлення реляцій після нормалізації. Завдяки своїй компонентній архітектурі React ідеально підходить для розробки складних користувацьких інтерфейсів, що складаються з компонентів, які можна використовувати повторно. Це було би корисним при створенні різних елементів, пов'язаних з нормалізацією бази даних: форм для введення реляцій, функціональних залежностей і компонентів для відображення результатів нормалізації.

Модель вводу/виводу Node.js - це неблокуючий цикл обробки подій, тобто коли викликається функція, яка виконує деякий ввід/вивід, вона не блокує виконання коду, а завершує його і дозволяє продовжити виконання інших частин вашого коду, поки операція вводу/виводу виконується у фоновому режимі. Це робить його особливо придатним для додатків, які вимагають інтенсивних операцій вводу/виводу, таких як читання з бази даних і запис до неї. Ця функція

гарантує, що система залишається швидкою навіть під навантаженням обробки складної логіки нормалізації. [10]

Крім того, на JavaScript розроблені фреймворки React і Node.js, що спрощує процес full-stack розробки, оскільки не потрібно постійно «перемикатися» між двома-трьома мовами або більше, що використовуються в технічному стеку.

3.2 Реалізація ключових операцій алгоритму

Як було згадано в попередньому розділі, при розробці системи автоматичної нормалізації структур баз даних Node.js надає зручний функціонал для ефективного розв'язання алгоритмів складних обчислювальних задач. У цьому підрозділі описується архітектура основного коду, реалізованого на Node.js, а акцент буде зроблено на базових класах, призначених для інкапсуляції основних елементів нормалізації: реляцій, функціональних залежностей та інших допоміжних структур для обробки і зберігання даних.

Основна частина коду, яка створює необхідні об'єкти, складається з декількох класів: Relation, Dependency, NormalizationData та Data. Кожен клас призначений для обробки певної частини представлення схеми бази даних та операцій, які мають бути виконані в процесі нормалізації.

Клас Relation інкапсулює реляційні зв'язки бази даних. Він включає набір атрибутів, до яких додаються можливі квазіключі та залежності у вигляді приватних полів. Конструктор класу Relation перевіряє валідність наданих атрибутів - атрибути не повинні бути порожніми, правильно відформатованими як рядки, а їхній вміст повинен бути валідними символами, зазвичай великими літерами для узгодженості, хоча в кінці алгоритму він буде все одно перетворений на великі літери. Крім того, клас Relation включає геттери і сеттери для отримання і встановлення властивостей, таких як пов'язані залежності і квазіключі, що знову ж таки є дуже важливим аспектом для алгоритму нормалізації. Він допомагає переконатися, що кожне відношення збереже свою цілісність, і забезпечує узгоджений інтерфейс для керування властивостями

відношення. Ось фрагмент класу Relation, який ілюструє ініціалізацію об’єкту та базову перевірку:

```
class Relation {
    #quasikey = null
    #deps = null
    constructor (attributes) {
        try {
            if (!attributes) {
                throw new Error("The set of attributes must be provided")
            }
            if (!Array.isArray(attributes) || !attributes.length > 0) {
                throw new Error("Array of symbols must be given")
            }
            if (!this.#checkElements(attributes)) {
                throw new Error("Every element has to be a string letter")
            }
            else {
                this.attributes = attributes.map( (i) => (i.toUpperCase()))
            }
        } catch (e) {
            console.log(e)
        }
    }
```

Рис. 3.2.1. Скріншот з коду бекенду застосунку, де реалізовано перевірку правильності подання атрибутів для реляції

Така перевірка у конструкторі класу допомагає забезпечити безпомилкову обробку даних, щоби на клієнтську частину можна було передати інформацію про неправильне введення даних.

Клас Dependency в даній системі відіграє важливу роль, оскільки він інкапсулює клас функціональних залежностей, які є фундаментальними для процесу нормалізації. У реалізації, клас Dependency призначений для зберігання та управління зв'язками між атрибутами у стислому вигляді. Він починається з конструктора, який вимагає двох параметрів, x і y, які представляють визначальний і залежний атрибути. І x, і y повинні бути надані у вигляді масивів рядків, де кожен рядок представляє атрибут, залучений у залежність. Конструктор спочатку перевіряє, чи вказано ці параметри і чи є вони масивами зі змістом, якщо ні, він видає помилку, гарантуючи, що жоден екземпляр залежності

не може бути створений без коректних даних. Саме так виглядає конструктор цього класу:

```
class Dependency {
    constructor (x, y) {
        try {
            if (!x || !y) {
                throw new Error("Both x and y arrays must be provided")
            }

            if (!Array.isArray(x) || !x.length > 0) {
                throw new Error("Array of symbols must be given for determinant")
            }

            if (!Array.isArray(y) || !y.length > 0) {
                throw new Error("Array of strings must be given for dependence")
            }

            if (!this.#checkElements(x) || !this.#checkElements(y)) {
                throw new Error("Every element has to be a string letter")
            } else {
                this.x = x.map( (i) => (i.toUpperCase()))
                this.y = y.map( (i) => (i.toUpperCase()))
            }
        } catch (e) {
            console.log(e)
        }
    }
```

Рис. 3.2.2. Скріншот з коду бекенду застосунку, де реалізовано перевірку правильності подання наборів атрибутів для створення функціональної залежності

Після завершення початкових перевірок конструктор обробляє масиви x та y шляхом приведення кожного атрибута до його верхнього регістру, що в подальшому забезпечує узгодженість у всій програмі, а також буде корисним для порівнянь і обробки, які відбуваються пізніше у процесі алгоритму нормалізації. Після цього кроку конструктор зберігає оброблені масиви у змінних екземпляра, роблячи їх частиною стану об'єкта.

Крім того, клас Dependency містить методи, які покращують його зручність використання в системі. Метод toString, наприклад, надає чітке рядкове представлення залежності у форматі "X → Y". Це особливо корисно для

налагодження програми та відображення залежностей у зручному для користувача форматі. Реалізуючи цей метод, кожну залежність можна легко перетворити у читабельний формат, що полегшує роботу з залежностями під час процесу нормалізації.

Клас Data є повноцінним контейнером, корисним для подальшої розробки застосунку, бо він містить усі реляції та функціональні залежності схеми бази даних. Цей клас допомагає керувати різними компонентами схеми бази даних під час виконання різних операцій процесу нормалізації.

Клас Data структурований для роботи з колекціями об'єктів реляцій і залежностей, ініціалізується двома параметрами: масивами реляцій і залежностей, що містять екземпляри класів Relation і Dependency відповідно. Таке налаштування полегшує цілісний підхід для подальшої розробки алгоритму, де реляції та пов'язані з ними залежності зберігаються централізовано, і до них можна швидко і зручно отримати доступ. Ось як виглядає конструктор класу Data:

```
class Data {
    constructor (relations, dependencies) {
        this.relations = []
        for (const relation of relations) {
            this.relations.push(relation)
        }

        this.dependencies = []
        for (const dependence of dependencies) {
            this.dependencies.push(dependence)
        }
    }
```

Рис. 3.2.3. Скріншот з коду бекенду застосунку, де відбувається створення контейнеру для отриманих даних в виді масивів реляцій та залежностей

Клас NormalizedData інкапсулює весь процес нормалізації, інтегруючи ключові операції для визначення мінімального покриття, квазіключа та декомпозиції схем баз даних у визначену нормальну форму, а саме у цій програмі це - третя нормальна форма. Для розробки цього класу було обрано повний алгоритм зведення поданої множини функціональних залежностей до 3-ої нормальної форми, вивчений протягом курсу «Бази даних та інформаційні системи» на 2-му курсі. Клас NormalizedData використовує вищезгадані класи Data, Dependency і Relation для ефективнішої роботи з реляціями і функціональними залежностями.

По суті, код класу NormalizedData починається з ініціалізації екземпляром класу Data, який об'єднує всі реляції і залежності схеми бази даних в єдиний об’єкт. Таке початкове налаштування гарантує, що всі наступні операції матимуть швидкий та зручний доступ до отриманих даних.

Метод findMinimalCover є однією з перших критичних операцій. Він спрощує функціональні залежності, щоб кожна з них мала єдиний атрибут з

правого боку та усуває надлишкові залежності. Перше спрощення досягається за допомогою методу #singleAttributeRightPart, який налаштовує кожну залежність відповідним чином. Після цього метод систематично усуває надлишкові залежності за допомогою методів #removingRedundantDeps і #removingRedundantLeftDeps, а потім за умови того, що в множині ще існуватимуть надлишкові залежності, ці кроки будуть ще повторюватись. Знову ж таки, ці кроки призначені для перетворення набору залежностей до його найпростішої та найнеобхіднішої форми без втрати будь-якої невід'ємної частини даних. Цей отриманий масив потім буде передано до створеного об'єкту Data для подальшого зберігання, та стане основою для подальших задач нормалізації.

```
findMinimalCover() {
    this.#singleAttributeRightPart()
    this.data.dependencies = this.#removeDuplicates(this.data.dependencies)
    this.#removingRedundantDeps()
    this.#removingRedundantLeftDeps()
    while (this.#changedLeft) {
        this.#removingRedundantDeps()
        this.#changedLeft = this.#checkMultipleLefts(this.data.dependencies)
        if (this.#changedLeft) this.#removingRedundantLeftDeps()
    }
}
```

Рис. 3.2.4. Скріншот з коду бекенду застосунку, де відбувається операція знаходження мінімального покриття множини функціональних залежностей з використанням створених приватних методів класу

Обчислення квазіключа, який є мінімальним набором атрибутів, що однозначно ідентифікує кожен кортеж у реляції, виконується методом getQuasikey. Він обчислює його, спочатку збираючи всі атрибути та ітеративно вилучаючи ті, які можуть бути визначені іншими на основі поточних залежностей, і потім потенційні квазіключі перевіряються за допомогою властивостей закриття, щоб підтвердити їхню правильність.

```
getQuasikey() {
    let candidates = this.#getAllAttrs().slice()
    const allAttrs = this.#getAllAttrs().slice()
    this.#allAttrs = [...allAttrs]
    for (const dep of this.data.dependencies) {
        for (const y of dep.y) {
            if (candidates.includes(y)) candidates.splice(candidates.indexOf(y), 1)
        }
    }

    let combos = this.#getAllCombos(candidates)
    let quasikey = null
    for (const key of combos) {
        const cl = this.#closure(new Dependency(key, allAttrs), this.data.dependencies)
        if (cl.sort().toString() === allAttrs.sort().toString()) {
            quasikey = key
            break
        }
    }
    return quasikey
}
```

Рис. 3.2.4. Скріншот з коду бекенду застосунку, де відбувається операція знаходження квазіключа

Декомпозиція схеми бази даних на 3NF виконується за допомогою методу get3NF. Спочатку він знаходить квазіключ за допомогою методу, описаного вище, оскільки без цього процес нормалізації буде некоректним. Потім він використовує мінімальне покриття залежностей, що зберігається в об'єкті Data, і список атрибутів для декомпозиції схеми. Фактична логіка декомпозиції інкапсульована у методі #decomposeTo3NF, який викликається в кінці методу get3NF.

Закритий метод #decomposeTo3NF класу NormalizedData призначений для перетворення схеми бази даних до третьої нормальної форми, метою якої є зменшення надмірності та усунення транзитивних залежностей, які не включають квазіключ. При цьому систематично застосовуються правила третьої нормальної форми до кожної реляції і всіх залежностей, щоб досягти схеми, яка балансує між цілісністю даних і ефективністю запитів.

Процес починається з перевірки кожної залежності в межах заданого мінімального покриття, щоб перевірити, чи відповідає схема другій і третій

нормальним формам. Якщо залежності реляції вже знаходяться в цих формах, метод просто повертає реляцію яка вона є, тому що подальша декомпозиція не потрібна. Однак, він переходить до більш детальної перевірки, якщо є порушення якоїсь нормальної форми, і тоді він декомпонує її.

Це означає, що неключовий атрибут частково залежить від підмножини складеного ключа, якщо ці залежності порушують другу нормальну форму. У таких випадках метод розділяє поточну реляцію на дві: одну з атрибутами залежності, а іншу - з рештою атрибутів, разом з ключовою частиною. Це розділення виконується обережно, щоб гарантувати, що отримана реляція не втратить жодних даних або залежностей, притаманних початковій схемі. Під час цього процесу декомпозиції метод використовує декілька перевірок, щоб гарантувати, що перетворення не зашкодять цілісності та повноті вихідної схеми. Ці перевірки включають перевірку того, чи потенційна декомпозиція призводить до об'єднання без втрат, тобто, якщо оригінальна реляція відновлюється з декомпонованої, то жодні дані не втрачаються. Для підтвердження того, що кожен крок декомпозиції відповідає цьому критерію, тут використовується теорема Хіта, яка перевіряється приватним методом.

Для залежностей, що порушують третю нормальну форму, коли неключовий атрибут транзитивно залежить від інших неключових атрибутів, метод використовує схожу стратегію. Він декомпонує реляцію таким чином, що залежні атрибути переміщуються в нову реляцію разом з атрибутами, від яких вони залежать. У кожній новій реляції цей крок також часто передбачає рекурсивну декомпозицію, оскільки кожна нова реляція оцінюється на предмет подальших порушень третьої нормальної форми.

Як бачимо, процес декомпозиції є ітеративним - для реляцій, які декомпонуються через порушення однієї з нормальних форм, кожна отримана реляція знову оцінюється для подальшої нормалізації. Так буде продовжуватися до тих пір, поки всі реляції не будуть відповідати вимогам третьої нормальної форми. Таким чином, це гарантує, що схема повністю нормалізована відповідно

до суворих правил цієї форми - небажана надмірність даних і потенційні аномалії оновлення ефективно усуваються.

Більше того, декомпозиція також обробляє всі залежності, прикріплені до кожної нової реляції. Залежності, які не мають відношення до декомпонованої реляції, видаляються, а нові залежності, що виникли в результаті декомпозиції, додаються. Таке управління залежностями є критично важливим для підтримки логічної узгодженості схеми бази даних.

Перевірки властивостей з’єднання без втрат та збереження залежностей теж реалізовані як окремі функції у даному класі, проте вони описані детально у 4 розділі, присвяченому тестуванню застосунку, оскільки є частиною перевірки роботи імплементованого алгоритму.

Бекенд системи даного застосунку, побудований за допомогою Node.js, призначений для обробки різноманітних запитів, пов'язаних з процесом нормалізації. Він використовує фреймворк Express.js для створення сервера, який відповідає на HTTP-запити, зокрема ті, що спрямовані на обчислення форм нормалізації, та перевірку таких властивостей, як з'єднання без втрат і збереження залежностей. Таке налаштування використовує можливості Node.js для обробки асинхронних операцій.

Сервер налаштовується за допомогою мінімального та гнучкого фреймворку для веб-застосунків, розроблених за допомогою Node.js – Express.js, який забезпечує надійні функції для програмних застосунків. Ініціалізація включає налаштування проміжного програмного забезпечення для роботи з CORS (Cross-Origin Resource Sharing), щоби забезпечити те, щоб сервер міг обробляти запити з різних доменів під час своєї роботи. Він також використовує вбудовану в Express.js функцію express.json() для отримання вхідних запитів за допомогою формату JSON, що досить зручно і корисно для читання даних, надісланих на сервер. Сервер прослуховує вказаний в коді порт (в даному випадку – 8888).

Внутрішня структура бекенду застосунку використовує маршрутизатори бібліотеки Express.js для ефективного управління ендпойнтами запитів. Маршрутизатор CalculationRouter, створений як окремий клас, використовується для групування всіх маршрутів запитів, пов'язаних з обчисленнями нормалізації, а саме з існуючими на клієнтській частин кнопками. Цей маршрутизатор прив'язується до маршруту «/calc». Відокремлення маршрутів для різних запитів не тільки краще організовує код, але й покращує його масштабованість.

Кожен маршрут у CalculationRouter налаштовано на обробку POST-запитів для різних типів операцій нормалізації:

- /min для обчислення мінімального покриття поданих функціональних залежностей
- /nf для обчислення декомпозиції третьої нормальної форми
- /lossless для перевірки властивості з'єднання без втрат
- /preserve для перевірки властивості збереження залежностей

Ці маршрути обробляються відповідними методами в CalculationController, які інкапсулюють логіку для кожного типу обчислень. Сам клас CalculationController містить методи, які безпосередньо взаємодіють з логікою нормалізації, інкапсульованою в класі NormalizedData. Кожен метод працює за схожим шаблоном:

- Отримуються необхідні дані з тіла запиту.
- Створюються екземпляри Relation і Dependency для кожної реляції і залежності, наданих у запиті.
- Створюється екземпляр NormalizedData зі створеними в попередньому пункті реляціями і залежностями.
- Виконуються обчислення процесу нормалізації за допомогою методів класу NormalizedData.
- Результати переводяться у формат, придатний для відповіді, і вони надсилаються на клієнтську сторону для відображення.

Обробка помилок є важливим аспектом кожного методу контролера, гарантуючи, що будь-які проблеми, пов'язані з вхідними даними або під час обробки, будуть виявлені, і відповідь про помилку буде надіслана назад і відображена у вигляді окремого повідомлення на клієнтській стороні застосунку.

3.3 Реалізація ключових операцій зі сторони клієнта

Як було згадано раніше, фронтенд застосунку було розроблено з використанням фреймворку React для швидкої зміни тексту результатів, що відображуються, та збирання даних з текстових полів. Певну увагу під час розробки було приділено веб-дизайну, щоби полегшити подальше користування програмою. Для більш зручної роботи зі сторінками на компоненти типу навігаційної панелі згори та основного контенту було додано темний синьо-фіолетовий градієнтний колір, а текст виділяється завдяки своєму білому контрастному кольору на більш темних компонентах та чорному на компонентах блідих відтінків.

При запуску застосунку спочатку відкривається головна сторінка, на якій описуються можливості роботи з програмою та кнопка переходу на сторінку програми, і за допомогою навігаційної панелі можна також перейти на сторінку програми або на іншу сторінку, де можна детальніше ознайомитись з алгоритмом.

Ось який вигляд має ця сторінка:

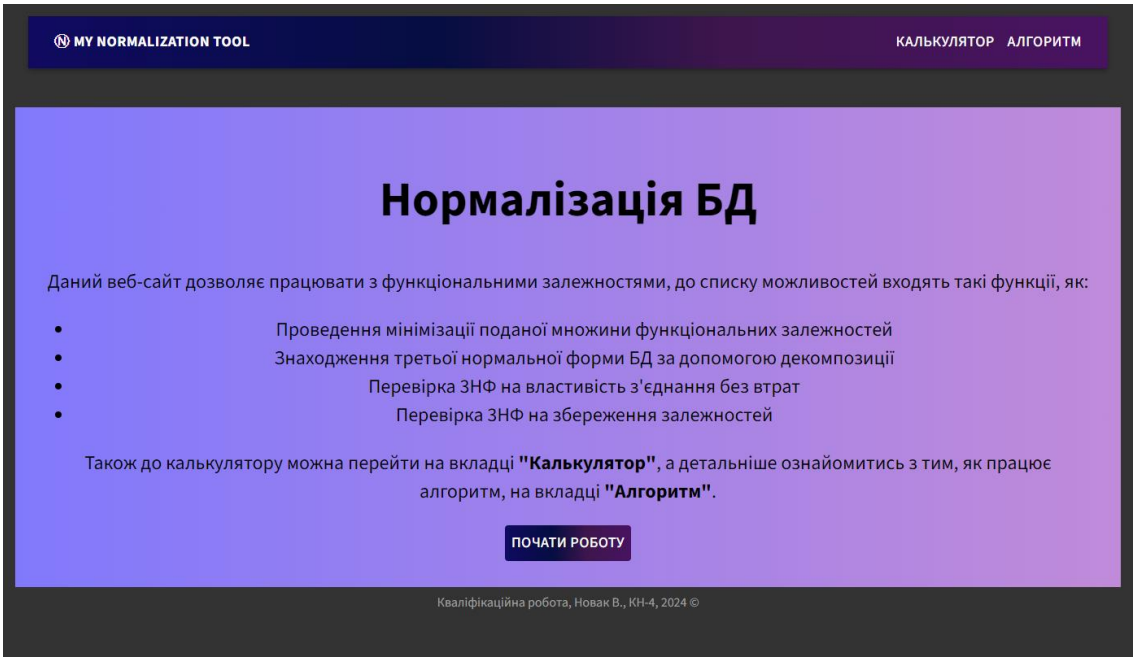


Рис. 3.3.1. Скріншот головної сторінки клієнтської частини застосунку

При переході на сторінку обчислень («Калькулятор») з’являється набір текстових полів, де перше велике поле призначене для введення усіх атрибутів реляції, а усі пари менших текстових полів – для введення функціональних залежностей. При додаванні нової залежності біля неї ж внизу з’являється чорний хрестик, на який можна натиснути, щоби видалити залежність, однак якщо на сторінці є лише одна залежність, її не можна видалити, бо немає внизу хрестику.

Дана сторінка з введеними даними виглядає ось так:

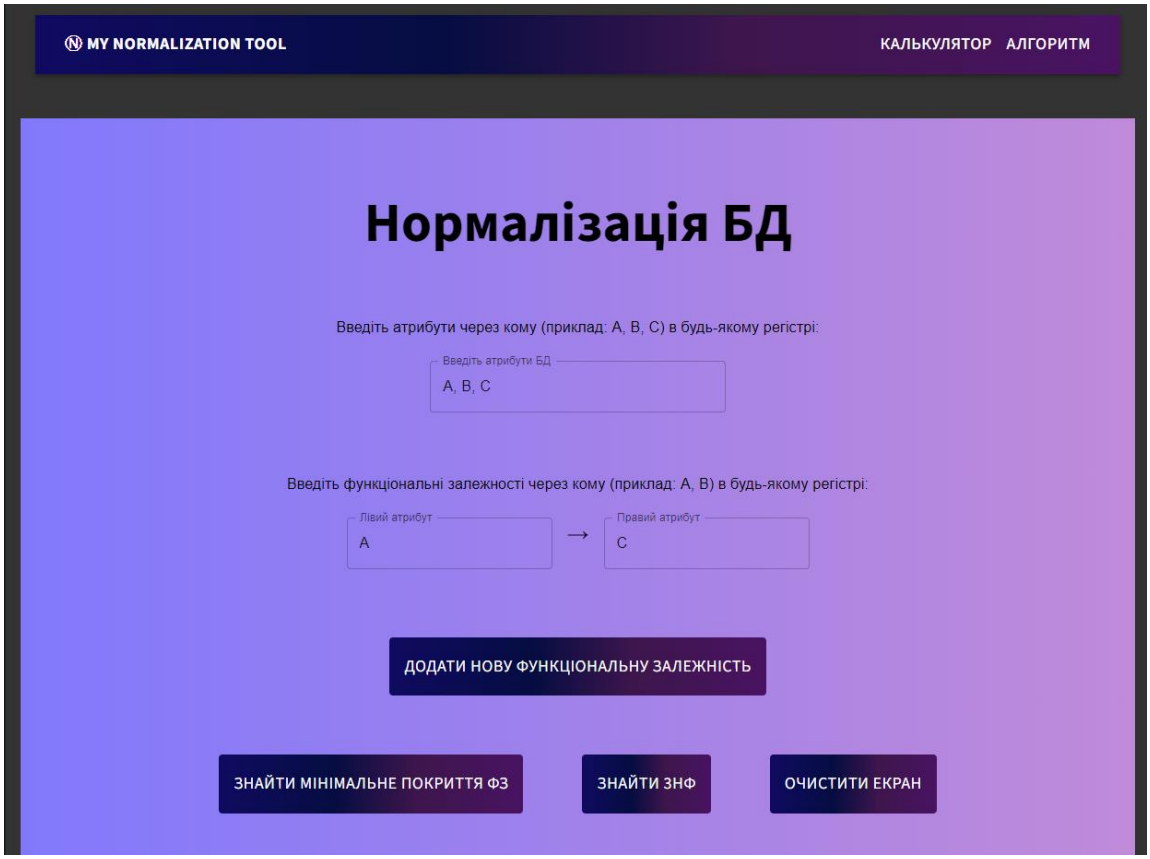


Рис. 3.3.2. Скріншот сторінки для обчислень основних операцій нормалізації

При натисненні на такі кнопки як «Знайти мінімальне покриття ФЗ», «Знайти 3НФ», «Перевірити 3НФ на властивість з’єднання без втрат» та «Перевірити 3НФ на властивість збереження залежностей» внизу в темній панелі з’являється відповідь на завдання, яке відповідає певній натисненій щойно кнопці, і це поле відповіді очищується при натисканні на кнопку «Очистити екран». Ось як виглядає отримана відповідь на сторінці:

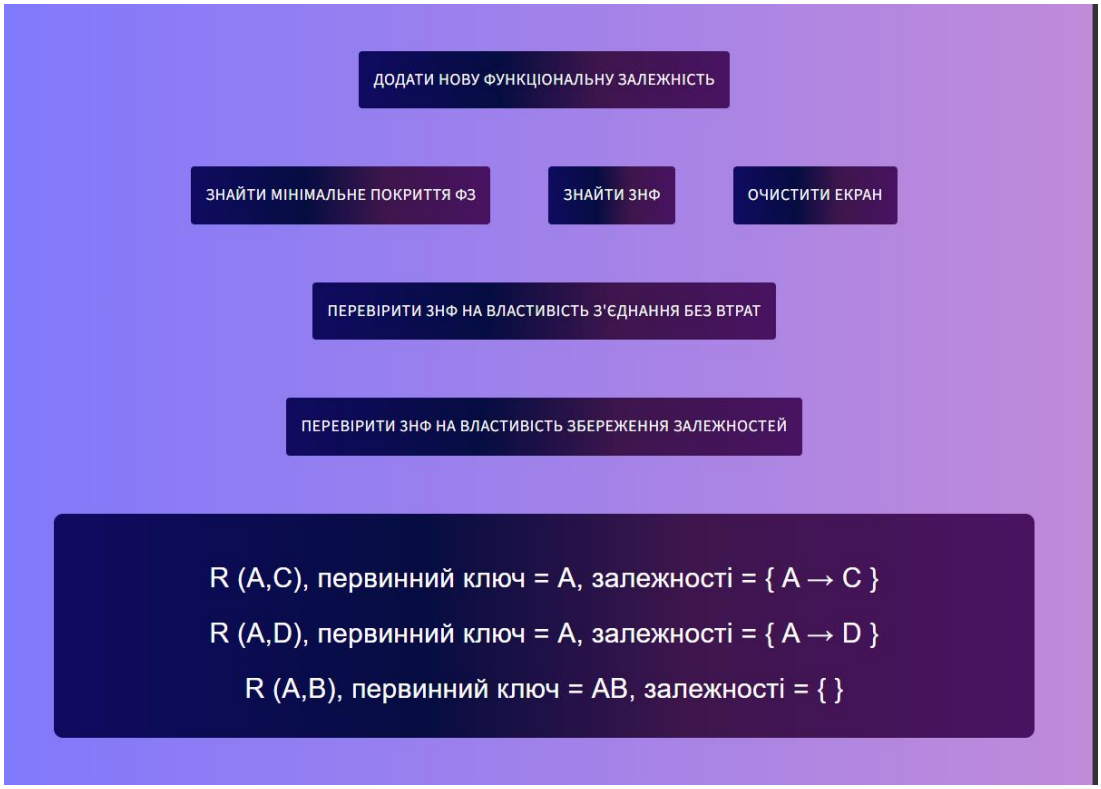
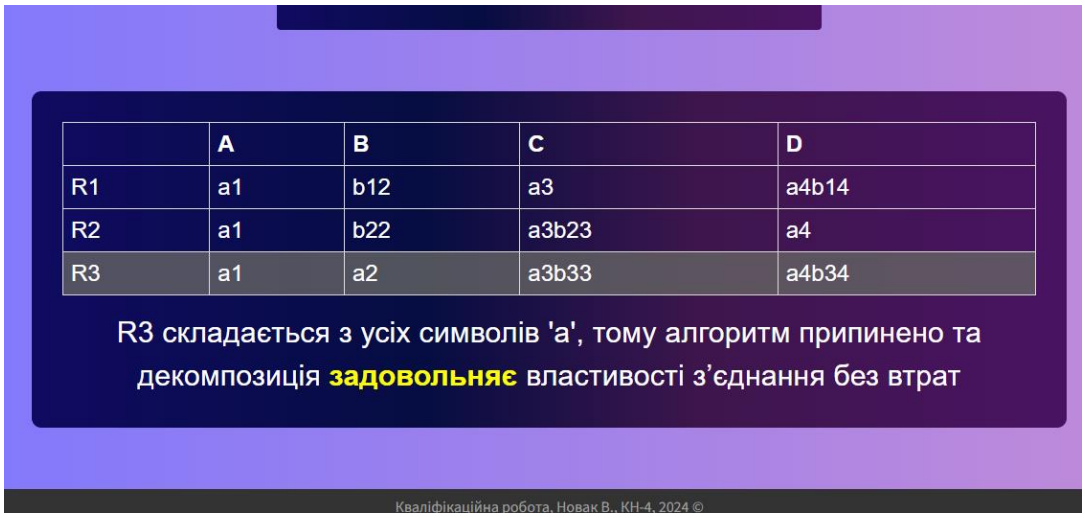


Рис. 3.3.3. Скріншот поля отриманої відповіді на сторінці для обчислень основних операцій нормалізації

Відповіді на різні обчислення подаються в нижню панель таким чином:

- Набір функціональних залежностей при натисненні на «Знайти мінімальне покриття ФЗ» (приклад: *«A → C, A → D, B → C»*)
- Набір реляцій при натисненні на «Знайти 3НФ» (приклад: «*R (A,C), первинний ключ = A, залежності = { A → C }; R (A,D), первинний ключ = A, залежності = { A → D }; R (A,B), первинний ключ = AB, залежності = { }*»)
- Таблиця, рядки якої відповідають отриманим у результаті поділу реляціям, а стовпці – аргументам початкової реляції, яка з'являється при натисненні на «Перевірити 3НФ на властивість з'єднання без втрат». Ось як виглядає така таблиця:



	A	B	C	D
R1	a1	b12	a3	a4b14
R2	a1	b22	a3b23	a4
R3	a1	a2	a3b33	a4b34

R3 складається з усіх символів 'a', тому алгоритм припинено та декомпозиція **задовольняє** властивості з'єднання без втрат

Рис. 3.3.4. Скріншот поля таблиці перевірки властивості з'єднання без втрат на сторінці для обчислень основних операцій нормалізації

- Результат об'єднання проекцій на реляції, та чи воно дорівнює множині всіх залежностей, тобто перевірка на властивість збереження залежностей при натисненні на відповідну кнопку (приклад:

Побудова проекції F_{min} на множини атрибутів:

$$R1\ (A,C),\ \text{первинний ключ} = A\ ,\ F1 = F[A,C] = \{A \rightarrow C\} \cup T$$

$$R2\ (A,D),\ \text{первинний ключ} = A\ ,\ F2 = F[A,D] = \{A \rightarrow D\} \cup T$$

$$R3\ (A,B),\ \text{первинний ключ} = AB$$

$$F^{+}_{min} \subseteq (\cup F)^{+} = (\{A \rightarrow C, A \rightarrow D\} \cup T)^{+} : \{B \rightarrow C\} \in (\cup F)^{+}\ ?$$

$$\{B \rightarrow C\}$$

$$\{B\}^{+} \cup = \{B\} \Rightarrow \{C\} \notin \{B\}^{+} \cup \Rightarrow \{B \rightarrow C\} \notin (\cup F)^{+}$$

Отже, властивість збереження залежностей ***не виконується****.*

Остання сторінка, на яку можна перейти при натисненні на надпис «Алгоритм» на навігаційній панелі згори, дозволяє детально переглянути роботу алгоритму на певному прикладі. Дана сторінка виглядає таким чином:

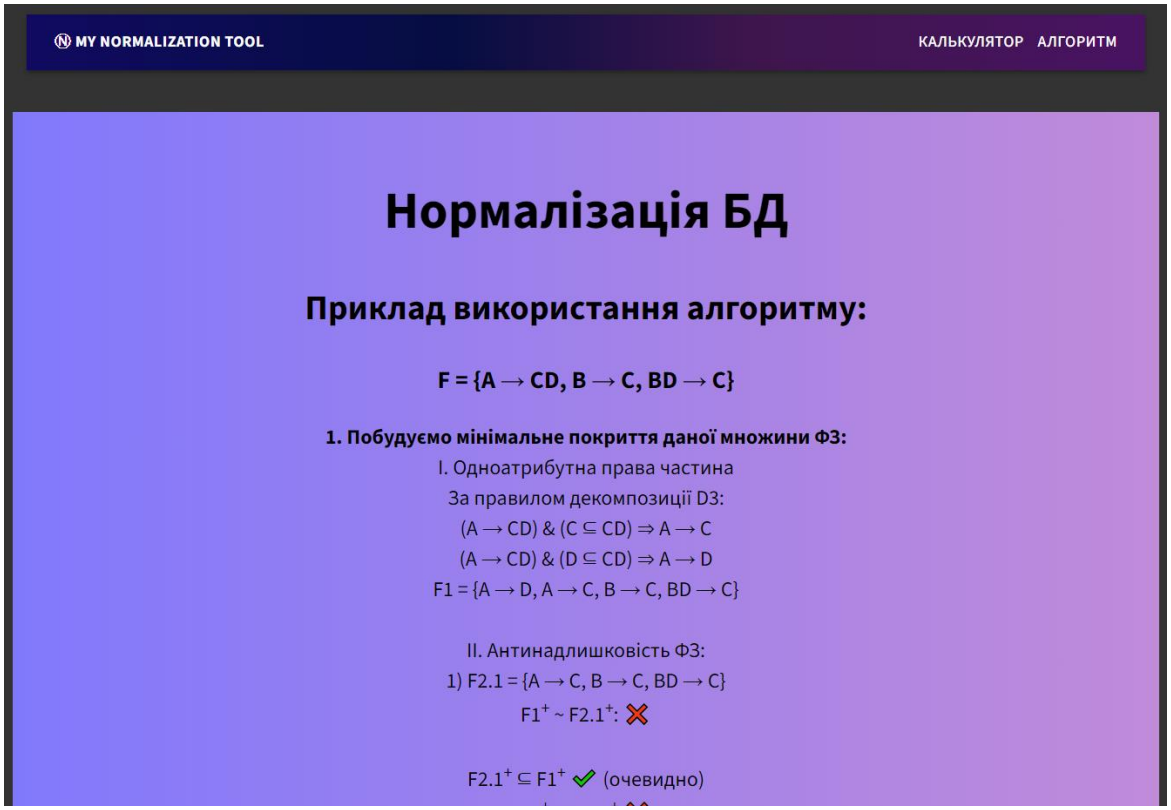


Рис. 3.3.5. Скріншот сторінки, на якій зображено приклад використання імплементованого алгоритму

Отже, з використанням такого технічного стеку, який включає фреймворки React.js, Node.js та Express.js, було успішно розроблено повністю функціонуючий веб-застосунок, який дозволяє знайти мінімальне покриття поданої множини функціональних залежностей, декомпонувати їх у реляції третьої нормальної форми, а також перевірити властивості з’єднання без втрат та збереження залежностей. Застосунок складається з протестованого та детально розробленого бекенду, та з інтуїтивного та швидко реагуючого на зміни в вхідних даних фронтенду.

РОЗДІЛ 4: ТЕСТУВАННЯ РОЗРОБЛЕНОЇ СИСТЕМИ НОРМАЛІЗАЦІЇ

4.1 Автоматизоване тестування програми

Вибір фреймворку для тестування та підхід до автоматизованого тестування також є питанням, вартим розгляду, при розробці надійного та якісного програмного забезпечення. Для даної системи нормалізації було обрано Jest як основний фреймворк для тестування. На це рішення вплинуло декілька факторів, що роблять Jest особливо придатним для тестування Node.js додатків, а саме тих тих, що включають складні логічні операції, включаючи ті, що зустрічаються при нормалізації баз даних.

Jest відомий своєю простотою та ефективністю, завдяки чому можна розробти автоматичне тестування доволі швидко, сприяючи швидкому налагодженню проекту. Однією з основних причин вибору Jest є його вбудована підтримка середовищ JavaScript та Node.js, що дозволяє легко інтегрувати його у вже існуючу структуру проекту без потреби в якомусь складному налаштуванні. Крім того, Jest надає повний набір функцій, які мають вирішальне значення для надійного тестування, таких як вбудовані assert-и, підтримка асинхронного тестування тощо. [11]

Процес тестування був побудований навколо перевірки кожного основного компонента бекенду системи нормалізації, щоби гарантувати те, що кожна функція працює належним чином за різних умов. Основні компоненти, що тестувалися, включають класи Relation, Dependency, Data і NormalizedData.

Кожен клас і метод був підданий unit-тестуванню, щоб переконатися, що усі вони працюють коректно. Наприклад, були написані тести для перевірки того, що клас Relation викидає помилки, коли ініціюється з невалідними даними, і перевірка того, що він правильно аналізує та зберігає інформацію про атрибути. Ось як виглядає цей код тесту:

```
import Relation from '../program/Relation.js'

describe('Relation class tests', () => {
    test('should create an instance with uppercase attributes', () => {
        const attributes = ['a', 'b', 'c'];
        const relation = new Relation(attributes);
        expect(relation.attributes).toEqual(['A', 'B', 'C']);
    });

    test('should throw error if attributes are not provided', () => {
        expect(() => new Relation()).toThrow('The set of attributes must be provided');
    });

    test('should throw error if attributes are not an array', () => {
        expect(() => new Relation('abc')).toThrow('Array of symbols must be given');
    });

    test('should throw error if any attribute is not a single letter string', () => {
        expect(() => new Relation(['A1', 'B', 'C'])).toThrow('Every element has to be a string letter');
    });

    test('should correctly set and get quasikey and dependencies', () => {
        const relation = new Relation(['a', 'b']);
        relation.setQuasikey(['A']);
        relation.setDeps(['A → B']);
        expect(relation.getQuasikey()).toEqual(['A']);
        expect(relation.getDeps()).toEqual(['A → B']);
    });
});
```

Рис. 4.1.1. Код unit-тестування класу Relation

Такий підхід до автоматизованого тестування за допомогою фреймворку Jest не тільки гарантує те, що кожна частина бекенду системи функціонує правильно, але й також те, що застосунок в цілому може надійно виконувати завдання нормалізації в різних сценаріях. Автоматизовані unit-тести призначені для повторного запуску після внесення змін до кодової бази, що забезпечує безперервну перевірку та підтримку якості коду протягом усього процесу розробки. Отже, таке тестування підтримує розробку надійної системи нормалізації, якій можна довіряти за точність і ефективність.

4.2 Тестування роботи алгоритму

Через те, що не існує спеціальних JavaScript-бібліотек для роботи з алгоритмами нормалізації структури бази даних, відповідно, перевірити процес роботи безпосередньо алгоритмів більш автоматизовано не вийде, тому потрібно застосовувати ручні методи перевірки. В першу чергу, для перевірки роботи алгоритму нормалізації в застосунку надзвичайно корисними були приклади завдань на нормалізацію поданих функціональних залежностей, пройдених

протягом курсу «Бази даних та інформаційні системи» на 2-му році навчання. До того ж, ці приклади наборів функціональних залежностей було внесено до онлайн-інструментів, розглянутих у підрозділі 2.1, і співставлено з результатами, отриманими у даному застосунку, і вони усі збігались між собою.

Ще одним способом перевірки правильності роботи алгоритму є практика перевірки реляцій на дотримання властивостей з'єднання без втрат та збереження залежностей. Для цих властивостей були створені відповідні функції в файлі NormalizedData.js, і ба більше, результати цих функцій відображаються при натисненні на відповідні кнопки у фронтенді застосунку, тому навіть третій користувач може запевнитись у правильності роботи алгоритму.

Функція checkLosslessCon() у класі NormalizedData призначена для перевірки властивості з'єднання без втрат при декомпозиції у схемі створеної реляційної бази даних. Функція створює двовимірний масив test, де кожен рядок відповідає реляції, а кожен стовпець - атрибуту схеми бази даних. Перший рядок і перший стовпець використовуються для міток: мітки першого рядка - це позначення атрибутів, а мітки першого стовпця - ідентифікатори реляцій (у форматі "R1", "R2" …).

Суть функції полягає в заповненні цієї матриці спеціальними маркерами: "a" з наступним номером стовпчика для атрибутів, які належать до реляції, і "b" з наступним ідентифікатором рядка і стовпчика для тих, які не належать. Це налаштування допомагає відстежувати додавання атрибутів у різні реляції. Потім під час проходження кожної залежності функція перевіряє, чи залежні атрибути у реляції все ще можуть бути визначені за допомогою атрибутів після декомпозиції, тоді вона позначає клітинки, що відповідають залежним атрибутам, а саме символом "a" з відповідним номером стовпчика для атрибутів, якщо всі визначальні атрибути присутні у реляції. Потім функція перевіряє, чи кожен атрибут у схемі може бути похідним принаймні від однієї реляції (тобто, позначається маркером "a" у кожному стовпчику впоперек рядків таблиці). Якщо ця умова виконується для хоч якоїсь залежності, функція підтверджує, що

декомпозиція виконана без втрат. На виході повертається таблиця, що забезпечує візуальне представлення зв'язків залежностей після декомпозиції. Для зручності передавання даних у фронтенд передається лише таблиця, і вже у фронтенді шукається рядок, у якому в усіх клітинках рядка є літера “a”, і виводиться відповідне речення про виконання або невиконання властивості.

Інша функція - checkDepsPreserving() у класі NormalizedData призначена для перевірки того, чи зберігається набір функціональних залежностей після декомпозиції схеми бази даних на декілька відношень.

Функція починається з визначення двох наборів залежностей: включені залежності, які зберігаються в новостворених зв'язках, і не включені залежності, які є вихідними залежностями з повної схеми, які не з'являються в жодному з нових зв'язків після декомпозиції. Для кожної з залежностей, що не увійшли до нових зв'язків (для цього є масив unincludedDeps), функція оцінює, чи можна вивести вплив цих залежностей із залежностей, що увійшли до нових зв'язків. Ця оцінка виконується за допомогою методу для знаходження замикання, який, вочевидь, обчислює замикання набору атрибутів щодо заданого набору залежностей, по суті визначаючи всі атрибути, які можуть бути функціонально визначені з початкового набору згідно з поточними залежностями. Функція вказує кожну залежність і перевіряє, чи замикання визначальних атрибутів залежності, обчислене з використанням лише включених залежностей, містить залежні атрибути залежності. Якщо це так для всіх таких залежностей, то це означає, що вихідні залежності збережено, навіть якщо вони не представлені явно в жодному новому відношенні.

Як і з попереднім методом, результати цих перевірок, разом із самими залежностями, зберігаються у масиві рішень, який також містить прапорці, що вказують на те, чи збережено кожну залежність. Якщо всі залежності збережено або немає жодного елемента в unincludedDeps, функція додає остаточний результат, який вказує на повне збереження, і потім знову у самому фронтенді

застосунку, дивлячись на цей результат, виводиться відповідне значення про виконання або невиконання властивості.

Отже, методологія тестування даного застосунку полягає в тому, щоб забезпечити коректну роботу кожного компонента системи за різних сценаріїв. Для основних компонентів, таких як класи Relation, Dependency, Data та NormalizedData, були побудовані unit-тести для підтвердження їхньої операційної цілісності та надійності. До того ж, до тестування програми було додано перевірку на виконання властивостей з’єднання без втрат та збереження властивостей, і ручна перевірка на відповідність вже існуючим прикладам схожих завдань.

ВИСНОВКИ

Отже, дана дипломна робота присвячена розробці автоматичної системи нормалізації структури баз даних, що пояснюється малою кількістю схожих інструментів в Інтернеті та в системах управління базами даних. Оскільки реляційні бази даних залишаються домінуючими завдяки своїй надійності та здатності обробляти структуровані запити, потреба в якісній нормалізації для підвищення продуктивності та цілісності баз даних досі існує.

Робота описує базові теоретичні аспекти алгоритму нормалізації, аналізуючи існуючі методи та їх обмеження в рамках сучасних систем управління базами даних. На основі цих знань, описаних у перших двох розділах, було створено практичну реалізацію автоматичної системи нормалізації з використанням сучасних фреймворків Node.js та React.js, які підтримують обробку даних у реальному часі та можливості динамічної зміни інтерфейсу користувача. Ця система виконує такі базові операції, як пошук мінімального замикання поданих функціональних залежностей, декомпозиція до реляцій третьої нормальної форми, перевірка властивостей з’єднань без втрат та збереження залежностей.

Автоматизоване тестування, що включає unit-тестування з використанням фреймворку Jest, та ручна перевірка функціональності алгоритмів гарантують надійність та коректність роботи застосунку за різних умов. Ці тести охоплювали окремі класи та методи, що мають важливе значення для бекенду системи, забезпечуючи надійну продуктивність та коректну функціональність. Практичні застосування, продемонстровані в роботі, підтверджують можливості застосунку ефективно нормалізувати структури бази даних.

Таким чином, ця робота не тільки надає функціональний інструмент для управління базами даних, але й демонструє власне дослідження алгоритмів нормалізації. Загалом, підкреслюється важливість інтеграції теоретичних знань з

практичними програмними рішеннями для покращення проектування та експлуатації баз даних.

Даний застосунок міг би стати у нагоді для детального вивчення теми нормалізації під час проходження курсів з баз даних та інформаційних систем.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ ТА ЕЛЕКТРОННИХ РЕСУРСІВ

1. Date C. J. An Introduction to Database Systems / Date., 2005. № 8. С. 437 - 440.
2. Connolly T. M. DATABASE SYSTEMS: A Practical Approach to Design, Implementation, and Management / T. M. Connolly, C. E. Begg., 2003. – № 3. С. 459–474
3. Пасічник В. В. Організація баз даних та знань / В. В. Пасічник, В. А. Резніченко. – Київ: Видавнича група BNV, 2006. – С. 137–149
4. Ziawasch A. DFD: Efficient Functional Dependency Discovery / Ziawasch Abedjan, Patrick Schulze, Felix Naumann, 2014. [Електронний ресурс] – Режим доступу до ресурсу: https://www.researchgate.net/publication/274711196_DFDEfficient_Discovery_of_Functional_Dependencies
5. Data & Database Normalization Explained: How To Normalize Data [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://www.splunk.com/en_us/blog/learn/data-normalization.html
6. Functional Dependencies Checker [Електронний ресурс] – Режим доступу до ресурсу: https://arjo129.github.io/functionalDependencyCalculator/
7. Relational Database Tools [Електронний ресурс] – Режим доступу до ресурсу: http://raymondcho.net/RelationalDatabaseTools/RelationalDatabaseTools.html
8. Boyce-Codd relation solver [Електронний ресурс] – Режим доступу до ресурсу: https://gamgi.github.io/js-bcnf/
9. Rout P. Database Normalization in SQL Server [Електронний ресурс] / Pranaya Rout – Режим доступу до ресурсу: https://dotnettutorials.net/lesson/database-normalization-sql-server/

10. Overview of Blocking vs Non-Blocking [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/en/learn/asynchronous-work/overview-of-blocking-vs-non-blocking
11. Jest Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://jestjs.io/uk/docs/getting-started