Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження моделей і архітектурних рішень для створення баз даних
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-6

Штельма А.С.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник к.т.н., доцент. Чуприна А.С.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

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

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Штельмі Антону Сергійовичу
(прізвище, ім'я, по батькові)

1.Тема роботи «Дослідження моделей і архітектурних рішень для створення баз даних»

Затверджена наказом по університету від 29.03.2024р. № 250 Ст

2.Термін подання студентом роботи до екзаменаційної комісії 01.06.2024

3.Вихідні дані до роботи інформація стосовно баз даних мереж, Data Base, інформація стосовно архітектур баз даних, моделі даних баз даних, мова програмування Python, фрєймворк Django, кластеризація баз даних , методи стиснення баз даних.

4.Перелік питань, що потрібно опрацювати в роботі

аналіз існуючих баз даних, найпоширеніші моделі даних та архітектурні рішення в базах даних їх переваги та недоліки, проведення експериментальних досліджень, написання програмних рішень, проведення експериментів та аналіз отриманих результатів, формулювання висновків стосовно реляційної бази даних, можливості використання шляхом поєднання кластерів та методів стиснення для реляційної бази даних. .

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз предметної галузі	29.03.2024	*виконано*
2	Виявлення проблематики галузі	30.03.2024 - 04.04.2024	*виконано*
3	Аналіз моделей даних	05.04.2024 - 10.04.2024	*виконано*
4	Аналіз архітектур баз даних	11.04.202 - 12.04.2024	*виконано*
5	Здійснення огляду фреймворку Django	12.04.2024 - 14.04.2024	*виконано*
6	Здійснення огляду бібліотек Python для наукового експерименту.	15.04.2024 - 17.04.2024	*виконано*
7	Здійснення огляду алгоритмів кластерів, та стиснення	18.04.2024 - 19.04.2024	*виконано*
8	Розробка Web застосунку у фреймвоку Django, та наповнення реляційної бази даних	19.04.2024 - 22.04.2024	*виконано*
9	Розробка алгоритму кластеризації та стиснення	22.04.2024 - 24.04.2024	*виконано*
10	Аналіз експерименту	24.04.2024	*виконано*
11	Написання пояснювальної записки	23.04.2024 - 29.04.2024	*виконано*
12	Підготовка доповіді та презентації	29.04.2024 - 03.05.2024	*виконано*
13	Нормоконтроль	04.05.2024 - 05.05.2024	*виконано*
14	Рецензування	06.05.2024 - 07.05.2024	*виконано*
15	Здача роботи у електронний архів	09.05.2024	*виконано*
16	Попередній захист	19.06.2024	*виконано*
17	Отримання відзиву від керівника роботи	19.06.2024	*виконано*
18	Захист кваліфікаційної роботи	25.06.2024	*виконано*

Дата видачі завдання . 29..березня . 2024 р.

Студент _____ Штельма А.С.

(підпис)

Керівник роботи _____ доц. Чуприна А.С.

(підпис)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 106 с., 56 рис., 2 табл., 22 джерел.

БАЗА ДАНИХ, РЕЛЯЦІЙНА БАЗА ДАНИХ, КЛАСТЕРИЗАЦІЯ, СТИСНЕННЯ, АРХІТЕКТУРА БАЗ ДАНИХ, МОДЕЛІ ДАНИХ БАХ ДАНИХ.

Об’єктом дослідження є різноманітні моделі та архітектурні рішення баз даних.

Метою роботи є аналіз, порівняння та оцінка різних моделей і архітектурних підходів до проектування та реалізації баз даних. Це дослідження спрямоване на виявлення найефективніших і найбільш відповідних рішень для різних типів застосувань та умов експлуатації баз даних. Також у рамках цієї теми було досліджено методи кластеризації та методи стиснення за для підвищення працездатності бази даних.

В результаті роботи було досліджено най поширеніші моделі даних та архітектурні рішення баз даних. Також було розглянуто найпоширеніші кластеризатори баз даних, та методи стиснення, їх аналіз та порівняння, а також різноманітні поєднання їх у базі даних.

DATABASE, RELATIONAL DATABASE, CLUSTERING, COMPRESSION, DATABASE ARCHITECTURE, DATABASE DATA MODELS.

The object of research is various models and architectural solutions of databases.

The purpose of the study is to analyze, compare, and evaluate various models and architectural approaches to database design and implementation. This research is aimed at identifying the most effective and appropriate solutions for different types of applications and operating conditions of databases. Also, within this topic, clustering methods and compression methods were investigated to improve database performance.

As a result, the most common data models and architectural solutions for databases were studied. The most common database clusters and compression methods, their analysis and comparison, as well as various combinations of them in the database were also considered.

Заява щодо самостійного виконання кваліфікаційної роботи та можливості її публікації в електронному архіві відкритого доступу ElArKhNURE.

Я, Штельма Антон Сергійович, студент гр. ІПЗм-22-6, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження моделей і архітектурних рішень для створення баз даних», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

У сучасному інформаційному суспільстві бази даних відіграють ключову роль у зберіганні, управлінні та обробці великої кількості даних. Від структурованих реляційних баз даних до нереляційних і гібридних рішень, вибір моделі та архітектури бази даних значною мірою впливає на продуктивність, масштабованість і надійність системи. Різні моделі баз даних, такі як реляційні, документні, графові, ключ-значення та колонкові, розроблені для вирішення специфічних потреб і вимог різних застосунків. Кожна з цих моделей має свої унікальні переваги та недоліки, які варто ретельно зважувати при виборі оптимального рішення для конкретного завдання.

Архітектурні рішення, такі як розподілені системи, кластеризація, реплікація та використання хмарних технологій, додатково ускладнюють процес проектування баз даних, але й відкривають нові можливості для підвищення ефективності та надійності систем. Розуміння різних підходів до проектування баз даних, а також їхнього впливу на продуктивність і управління даними, є критичним для успішного впровадження інформаційних систем, що відповідають сучасним вимогам бізнесу та технологій.

Розподілені системи передбачають зберігання та обробку даних на кількох серверах або вузлах, що підвищує масштабованість і надійність системи. Такі рішення дозволяють розподіляти навантаження між кількома серверами і забезпечувати безперебійну роботу навіть у разі виходу з ладу одного з вузлів.

Кластеризація баз даних[16] передбачає об’єднання кількох серверів у кластер для забезпечення високої доступності та продуктивності. Кластерні бази даних автоматично розподіляють дані та запити між вузлами кластера, що дозволяє обробляти більше запитів одночасно і забезпечувати стійкість до відмов.

Реплікація даних полягає у створенні копій бази даних на кількох серверах. Це забезпечує високу доступність даних і захист від втрат у разі збою одного з

серверів. Реплікація може бути синхронною (з оновленням даних у реальному часі) або асинхронною (з затримкою в оновленні даних).

Використання хмарних технологій для зберігання та управління базами даних стає все більш популярним. Хмарні бази даних забезпечують гнучкість у масштабуванні, зниження витрат на інфраструктуру та високу доступність даних. Вони також пропонують різні моделі розгортання, такі як приватні, публічні та гібридні хмари, що дозволяє організаціям обирати оптимальні рішення відповідно до своїх потреб. Розуміння різних моделей та архітектурних рішень у сфері баз даних є критичним для вибору оптимальної стратегії зберігання та обробки даних. Реляційні, документні, графові, ключ-значення та колонкові бази даних мають свої унікальні характеристики, які визначають їхню придатність для різних типів застосувань. Архітектурні рішення, такі як розподілені системи, кластеризація, реплікація та хмарні технології, дозволяють підвищувати продуктивність, масштабованість та надійність баз даних. Обираючи підходящу модель та архітектуру бази даних, організації можуть забезпечити ефективне управління даними, що відповідає сучасним вимогам бізнесу та технологій, а також сприяти успішній реалізації своїх інформаційних систем.

1. ОПИС ПРОБЛЕМНОЇ ГАЛУЗІ

1.1 Аналіз проблемної області баз даних

Бази даних є невід'ємною частиною сучасних інформаційних систем, забезпечуючи зберігання, управління та доступ до даних. Однак, їх використання стикається з низкою проблем та викликів, які необхідно враховувати при проектуванні та експлуатації систем баз даних. Основні проблеми, з якими стикаються бази даних це масштабованість, продуктивність, узгодження та цілісність даних, безпека, відмовостійкість та резервне копіювання та останнім це гнучкість та адаптивність.

Почнемо з масштабованість адже зі зростанням обсягу даних та кількості користувачів виникає необхідність у масштабуванні баз даних. Буває вертикальне та горизонтальне масштабування.

Вертикальне масштабування[5] відбувається за рахунок додавання ресурсів до одного сервера (наприклад, збільшення обсягу оперативної пам'яті або потужності процесора). Це може бути дорогим і мати фізичні обмеження.

Горизонтальне масштабування відбувається за рахунок додавання нових серверів до кластеру бази даних. Це вимагає складних архітектурних рішень і може призвести до проблем з узгодженістю даних.

Наступним є продуктивність адже забезпечення високої продуктивності баз даних є критично важливим, особливо для додатків з високими вимогами до швидкості обробки запитів. До продуктивності входять індексація та кешування

Використання індексів для прискорення [8] пошуку даних, але надмірна кількість індексів може уповільнити операції вставки, оновлення та видалення.

Кешування відбувається за рахунок Зберігання часто запитуваних даних у пам'яті для швидкого доступу. Необхідно враховувати час оновлення кешу і його узгодженість з основними даними.

Також забезпечення цілісності та узгодженості даних є ключовим завданням для будь-якої бази даних. До них входять транзакції та реплікатції.

Підтримка транзакцій з властивостями[9] ACID (атомарність, консистентність, ізоляція, довговічність) для забезпечення надійності операцій з даними.

Реплікацією називають синхронізацію даних між різними серверами або вузлами, що може призводити до проблем з узгодженістю у розподілених системах.

Безпека - захист даних від несанкціонованого доступу та забезпечення їхньої конфіденційності є критично важливими аспектами.

Аутентифікація та авторизація – забезпечує доступ до даних тільки авторизованим користувачам.

Шифрування або захист даних за допомогою шифрування як на рівні зберігання (дані на диску), так і на рівні передачі (дані, що передаються мережею).

Також є відмовостійкість та резервне копіювання, забезпечення безперервності роботи баз даних навіть у випадку збоїв є важливим для багатьох критично важливих систем.

Резервне копіювання відбувається шляхом регулярного створення резервних копій даних для відновлення у випадку втрат або пошкоджень.

Відновлення після збоїв відноситься де механізми автоматичного переключення на резервні сервери або вузли у випадку відмов основних компонентів.

Гнучкість та адаптивність це здатність баз даних легко адаптуватися до змін вимог та умов експлуатації. До яких входять зміна схеми даних та підтримка різних типів даних.

Зміна схеми даних це можливість змінювати структуру бази даних без значного впливу на роботу системи.

Підтримка різних типів даних це здатність зберігати та обробляти різні типи даних, такі як структуровані, напівструктуровані та неструктуровані дані.

Отже аналіз проблемної області баз даних виявляє ряд ключових викликів, з якими стикаються системи баз даних у сучасному світі. Масштабованість, продуктивність, узгодженість даних, безпека, відмовостійкість та гнучкість є основними аспектами, які необхідно враховувати при проектуванні, розробці та експлуатації баз даних. Розуміння та ефективне вирішення цих проблем дозволяє створювати надійні, ефективні та безпечні системи управління даними, які відповідають вимогам сучасних додатків та користувачів.

1.2 Аналіз аналогів моделей даних

У сучасному світі існує безліч систем управління базами даних (СУБД), кожна з яких має свої особливості, переваги та недоліки. Вибір конкретної моделі та архітектури бази даних залежить від конкретних вимог проекту, обсягів даних, типів даних та інших факторів. У цьому аналізі розглянуто популярні бази даних, їх моделі та архітектури.

1.2.1 Реляційна модель[10] даних вона заснована на поданні бази даних у вигляді таблиць, що складається з рядків та стовпців. Дані організовані у відносинах таблиці, де кожне відношення має набір атрибутів стовпців.

До переваг можна віднести це простоту і зрозумілість адже реляційна модель використовує таблиці зі стовпцями та рядками, що дозволяє легко розуміти структуру даних і проводити операції з ними. Наступним це гнучкість адже вона дозволяє легко вносити зміни у структуру даних без необхідності перепроектування всієї бази даних. Також незалежність даних від застосунків де дані можуть бути запитані та маніпульовані без залежності від конкретного застосунку, що сприяє більш широкому використанню та пере використанню даних. Цілісність даних реляційної моделі надає можливість встановлення

обмежень цілісності даних, таких як унікальність значень у стовпцях або зв'язки між таблицями, що допомагає у підтримці консистентності даних. Наступне це масштабованість адже реляційні бази даних можуть бути масштабовані як вертикально (додавання нових стовпців) так і горизонтально (додавання нових рядків) в залежності від потреб.

До недоліків[10] можна віднести обмежена представлення складних відносин де у деяких випадках складні взаємозв'язки між даними можуть бути складно виразити в реляційній моделі без перекомпонування даних або використання додаткових технік, що може призвести до складності та непродуктивності. Наступним є це накладні витрати на з'єднання адже при великій кількості таблиць і зв'язків між ними можуть виникати складності з ефективністю операцій з'єднання, особливо у великих базах даних. Також присутня неструктуровані дані адже реляційна модель не завжди ефективно працює з напівструктурованими або неструктурованими даними, які стають все більш поширеними в сучасному світі інформації. Висока накладна вартість на операції з даними де операції над даними у реляційній моделі можуть вимагати значних ресурсів обчислювальної потужності, особливо при роботі з великими обсягами даних.

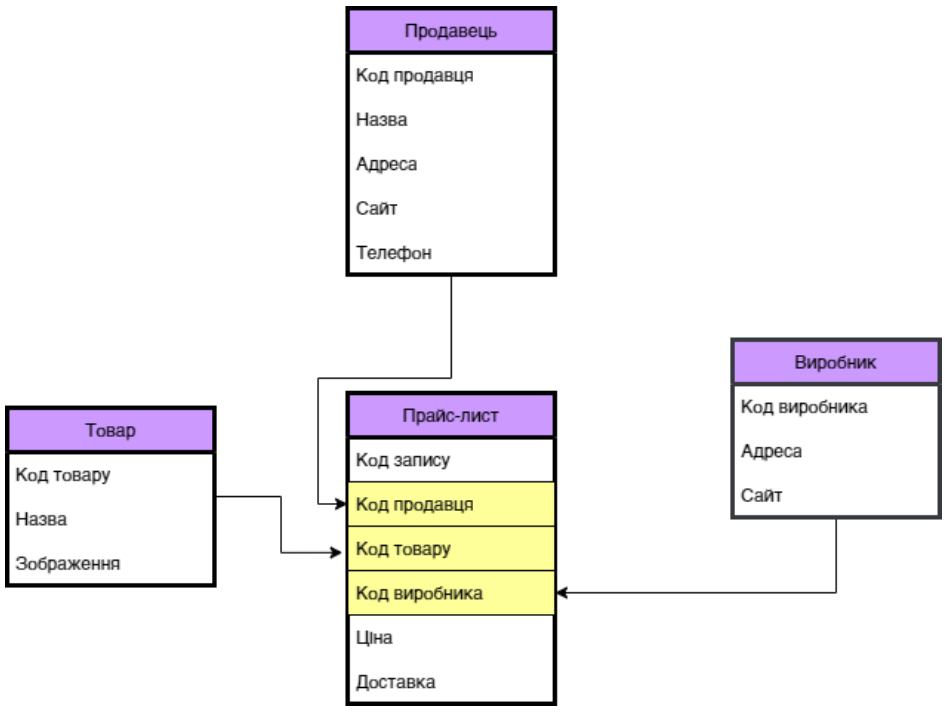


Рисунок 1.1 — Блок схема реляційна модель даних (виконано самостійно)

Реляційна БД [10]– це сукупність таблиць, пов'язаних між собою. Рядки таблиці містять дані про один об'єкт (наприклад, клієнта, користувача сайту, товар), а в стовпцях описуються характеристики цих об'єктів (наприклад, електронна адреса клієнта, код товару) як зображено на рисунку 1.1.

1.2.2 Ієрархічна модель даних[14] - це структура організації даних, в якій дані представлені у вигляді дерева зі зв'язками батько-дитина. Кожен запис у дереві може мати тільки одного батька, крім кореневого запису, який не має батька.

До переваг ієрархічної моделі[14] даних можна віднести це простоту структури адже модель даних є відносно простою для розуміння та використання. Вона відображає структуру даних у вигляді дерева, що може легко інтерпретуватися та оброблятися. Наступним є швидкий доступ до даних адже кожен запис має лише одного батька, доступ до даних може бути дуже швидким і ефективним, особливо якщо використовуються оптимізовані алгоритми для навігації по дереву. Також можна віднести підтримку багаторівневих відносин де модель даних дозволяє легко виражати складні відносини між даними, такі як багаторівневі структури департаментів у компанії або організаційні структури. Ефективність підтримка запитів з обмеженнями запитів, які стосуються конкретних гілок або частин дерева, можуть бути виконані досить швидко, оскільки структура даних сприяє швидкому доступу до цих даних.

До недоліків можна віднести обмеженість у виразності моделі даних, оскільки вона передбачає строго ієрархічну структуру. Це може ускладнити моделювання деяких типів даних, що мають більш складні або нелінійні відносини. Також присутні проблеми з реорганізацією даних, вони пов’язані зі змінами в структурі даних котрі можуть виявитися складними, оскільки будь-яка реорганізація може потребувати значних змін у вже існуючих зв'язках. До недоліків[22] також можна віднести проблеми з обробкою рекурсивних структур адже ієрархічна модель може мати проблеми з обробкою рекурсивних структур, де елемент може бути батьком або дитиною самому собі. Також присутня нестабільність даних при зміні структури, наприклад якщо структура даних

змінюється, це може призвести до нестабільності даних та важкостей у підтримці консистентності.

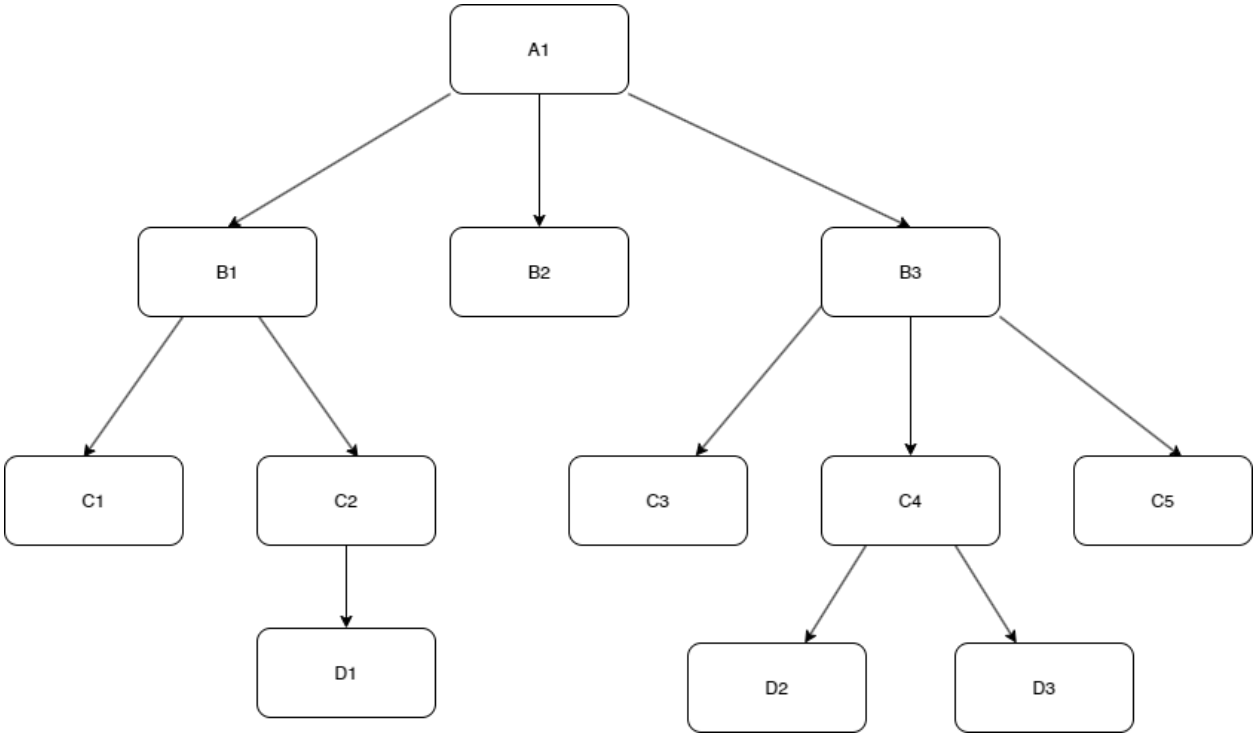


Рисунок 1.2 — Блок схема ієрархічна модель даних (виконано самостійно)

На блок схемі вижче зображено ієрархічна модель даних в базі даних

1.2.3 Мережева модель даних - це розширення ієрархічної моделі, де вузли можуть мати кілька батьків,[11] що дозволяє більш гнучко організовувати дані і встановлювати складні відносини між ними.

До переваг можна віднести це гнучкість виразності адже мережева модель дозволяє легко виражати складні відносини між об'єктами, оскільки вона дозволяє вузлам мати кілька батьків. Наступним є ефективність в обробці даних завдяки можливості вузлів може мати кілька батьків також можна легко отримувати та оновлювати дані, що знаходяться на різних рівнях ієрархії. До переваг також можна віднести це можливість моделювання складних відносин, де мережева модель дозволяє моделювати складні взаємозв'язки між об'єктами, які можуть бути складно виразити в інших моделях даних. Також присутня підтримка рекурсивних структур, завдяки якій мережева модель дозволяє легко обробляти рекурсивні структури даних, де об'єкт може бути батьком або дитиною самому собі.

До недоліків можна віднести складність структури даних тому що мережева модель може бути складною для розуміння та використання, оскільки потребує ретельного розуміння взаємозв'язків між вузлами. Присутні проблеми з обробкою даних в яких операції зчитування та запису даних можуть бути складними, оскільки вони можуть вимагати складних алгоритмів для навігації по мережі. Мережева модель також має високий ризик виникнення помилок, це трапляється завдяки складній структурі даних і можливості вузлів мати багато батьків, існує високий ризик виникнення помилок при роботі з даними. Виникають складності в підтримці цілісності даних котрі і в ієрархічній моделі, зміни в структурі даних можуть виявитися складними, оскільки вони можуть призвести до нестабільності даних та важкостей у підтримці консистентності.

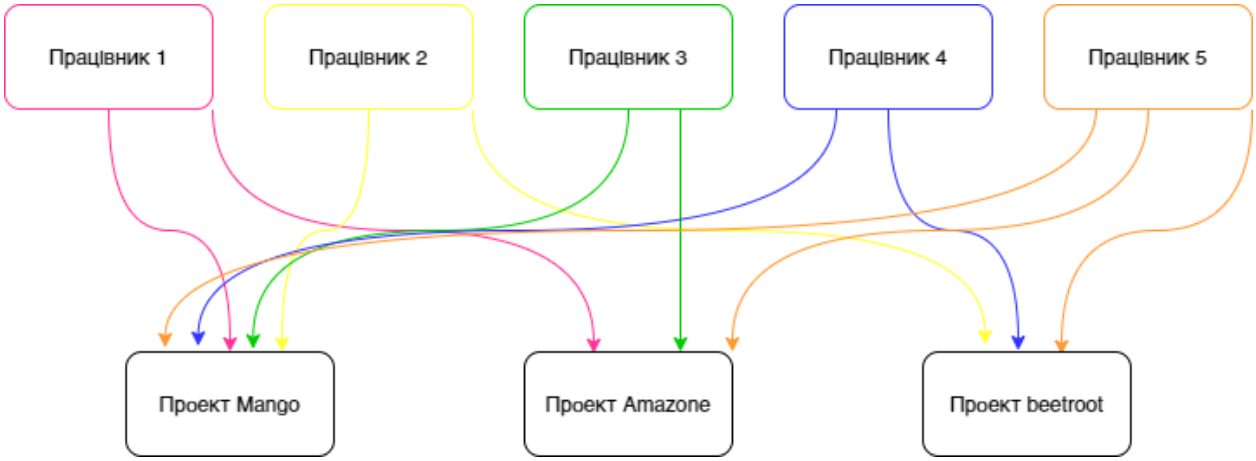


Рисунок 1.3 — Блок схема мережева модель даних (виконано самостійно)

1.2.4 Об'єктно-орієнтована модель даних (ООМД)[6] є способом організації даних, який базується на концепціях об'єктно-орієнтованого програмування (ООП). У цій моделі дані організовані у вигляді об'єктів, кожен з яких має атрибути (стани) і методи (поведінку).

До переваг можна віднести натуральність моделювання[3] тому що об'єктно-орієнтована модель даних дозволяє більш природньо відображати реальні об'єкти та їх взаємодії у програмах і системах. Наступним є модульність і перевикористання коду адже ООМД сприяє використанню модульних підходів та

перевикористанню коду завдяки упорядкуванню даних у вигляді об'єктів, які можуть бути легко використані в різних частинах програми. Наступним є спадкування та поліморфізм де один з ключових аспектів ООП - це можливість використання спадкування та поліморфізму, що дозволяє створювати ієрархії класів та збільшує гнучкість та повторне використання коду. Зручність у взаємодії з базами даних можна сюди додати адже деякі системи управління базами даних (СУБД) підтримують об'єктно-реляційні або об'єктно-орієнтовані бази даних, що дозволяє більш пряме відображення ООМД у базі даних. Наступним є легкість розробки та підтримки адже ООМД спрощує процеси розробки, тестування та підтримки програм завдяки більш прозорому та структурованому підходу до організації даних.

До недоліків можна віднести складність дизайну та реалізації адже побудова ефективних і гнучких об'єктно-орієнтованих моделей може вимагати значних зусиль та досвіду, особливо у великих та складних системах. Наступним є перевантаження об'єктами де у невірному використанні ООМД може призвести до перевантаження пам'яті та ресурсів обчислювальної системи через зайвий об'єктний оверхед. Також присутня складність оптимізації та швидкості. В деяких випадках оптимізація та підвищення швидкодії програм, побудованих на основі ООМД, може бути складною через велику кількість об'єктів та їх взаємодію. Не всі бази даних підтримують ООМД, деякі СУБД[22] не підтримують об'єктно-орієнтовані можливості або реалізовані недостатньо ефективно, що може обмежувати використання ООМД в певних проектах.

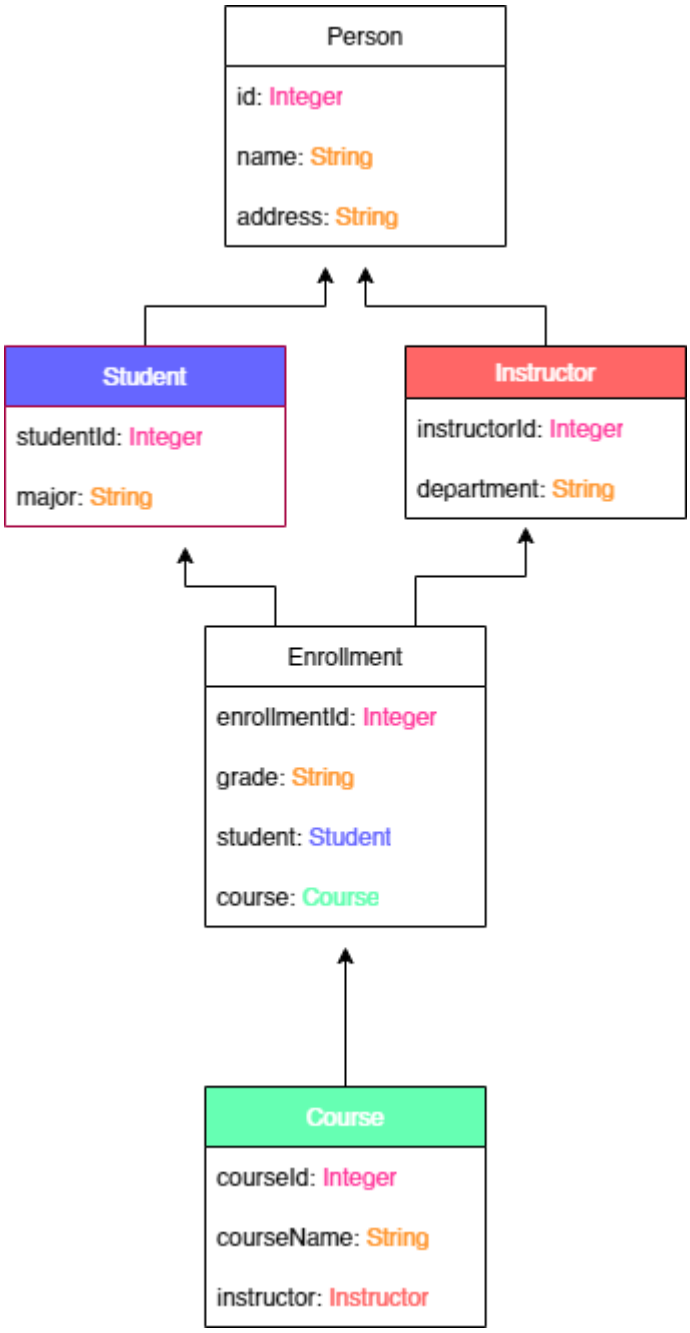


Рисунок 1.4 — Блок схема об’єктно-орієнтована модель даних (виконано самостійно)

Розглянемо приклад системи управління студентами та курсами[22], де студенти записуються на курси. Ця блок схема має декілька класів:

“Person” - базовий клас для всіх людей;

“Student” - клас для студентів, який наслідує від класу Person;

“Instructor” - клас для викладачів, який також наслідує від класу Person;

“Course” - клас для курсів;

“Enrollment” - клас для запису студентів на курси.

Student та course мають зв'язок багато-до-багатьох через об'єкт enrollment. instructor викладає один або кілька course. enrollment містить інформацію про запис студента на курс, зокрема посилання на об'єкти atudent та course.

1.2.5 Графова модель даних[10] - це спосіб організації даних, в якому дані представлені у вигляді графа, що складається з вершин (вузлів) та ребер (зв'язків) між ними. Кожна вершина представляє об'єкт, а кожне ребро показує зв'язок між об'єктами.

До переваги відноситься гнучкість у вираженні взаємозв'язків. Графова модель дозволяє легко виражати складні взаємозв'язки між об'єктами[14], що робить її ефективною для моделювання різноманітних систем і взаємодій. Також до ефективності у виконанні запитів на зв'язки, запити, пов'язані з взаємозв'язками між об'єктами, можуть бути виконані швидко та ефективно завдяки структурі графа. Простота аналізу структури даних адже графова модель надає зручний спосіб для візуалізації та аналізу структури даних, що допомагає зрозуміти їхні взаємозв'язки. Можливість моделювання різних типів взаємозв'язків, завдяки якій дозволяє моделювати різні типи взаємозв'язків, такі як взаємодії, завдання, залежності та інші, що робить її корисною для широкого спектру застосувань.

До недоліків можна віднести складність у виразності деяких запитів адже деякі запити, особливо ті, що стосуються[13] глибоких або складних взаємозв'язків, можуть бути складними для формулювання та виконання в графовій моделі. також можна віднести це потенційну високу складність структури графа, адже у деяких випадках структура графа може стати дуже складною, особливо великими та густонаселеними графами, що може призвести до проблем з її аналізом та оптимізацією. Присутня також потреба у високій обчислювальній потужності тому що для обробки великих та складних графів може знадобитися значна обчислювальна потужність та ресурси, що може стати проблемою у великих

системах. Також не всі типи даних підходять для графової моделі. У деяких типів даних можуть бути менш ефективними для виразу в графовій моделі, зокрема, якщо вони не мають явних зв'язків між собою.

Отже в цілому, графова модель даних є потужним інструментом для моделювання та аналізу взаємозв'язків між об'єктами, але вона також має свої виклики та обмеження, які потрібно враховувати при її використанні.

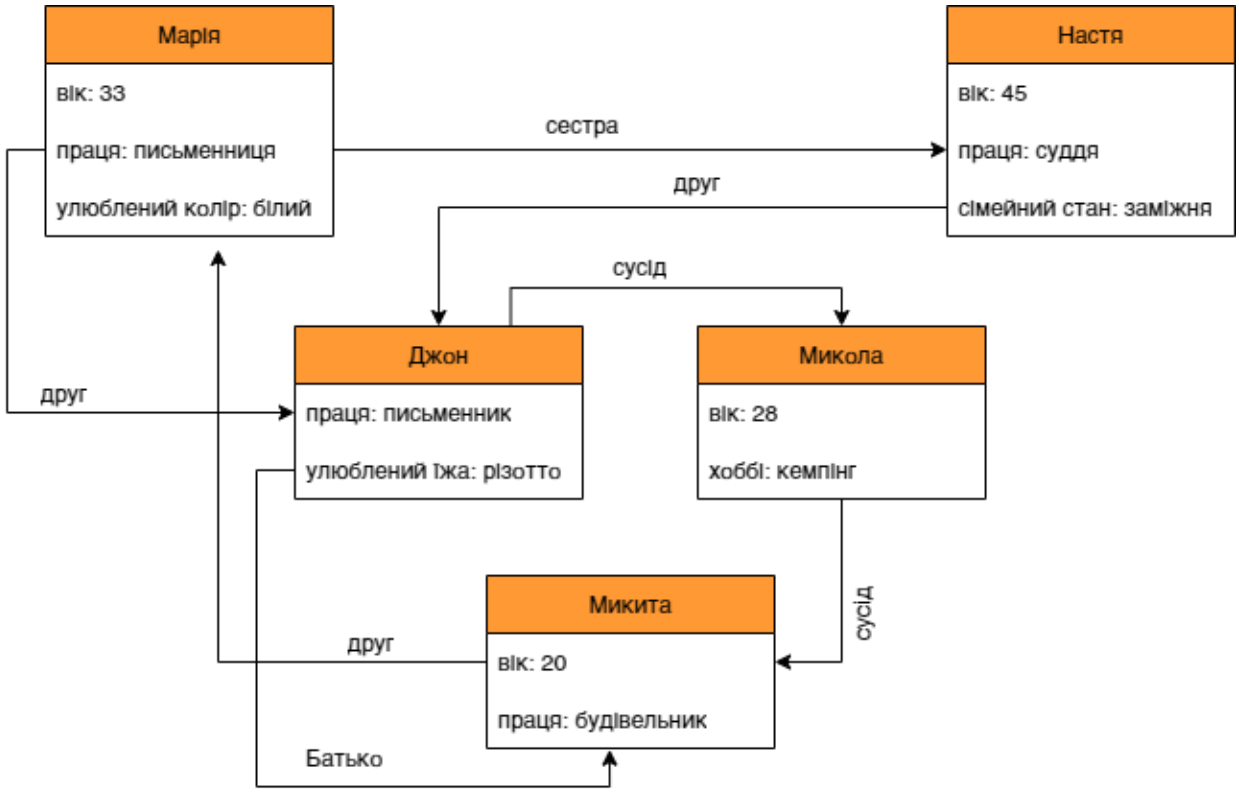


Рисунок 1.5 — Блок схема графова модель даних (виконано самостійно)

1.2.6 Документно-орієнтована модель даних (ДОМД)[22] є типом моделі даних, в якій дані організовані у вигляді документів, що зазвичай представлені у форматах, таких як JSON (JavaScript Object Notation) або XML (eXtensible Markup Language). Кожен документ може містити вкладені структури даних, такі як об'єкти або масиви, і може бути незалежним від інших документів.

До переваг можна віднести гнучкість у структурі даних адже вона дозволяє зберігати даний будь-якої структури в кожному документі, що надає гнучкість у моделюванні різноманітних типів даних. Наступне це природність для обробки напівструктурованих даних ця модель даних ідеально підходить для зберігання та

обробки напівструктурованих даних, які можуть мати змінну або невизначену структуру. Простота використання з мовами програмування та API[22], багато мов програмування мають вбудовану підтримку для роботи з форматами документно-орієнтованих даних, що полегшує їх обробку та обмін між різними системами. Горизонтальна масштабованість це коли документно-орієнтовані бази даних зазвичай добре масштабуються горизонтально, що дозволяє легко розширювати їх, додавши нові сервери для обробки додаткового обсягу даних.

До недоліків можна віднести менша ефективність для деяких типів запитів, адже документно-орієнтована модель може бути менш ефективною для деяких типів запитів, особливо тих, які потребують складних операцій з'єднання та агрегації даних. Наступним є можливість дублювання даних, у документно-орієнтованих базах даних може виникати проблема дублювання даних, оскільки інформація може бути повторювана у кількох документах. Присутня також складність у підтримці цілісності даних, оскільки документи можуть мати вкладені структури, складно забезпечити цілісність даних, особливо якщо вони змінюються часто або використовуються багатьма користувачами одночасно. Має також обмежену підтримку[14] операцій транзакцій, де деякі системи управління документно-орієнтованими базами даних можуть мати обмежену підтримку операцій транзакцій, що може стати проблемою у великих та високонавантажених системах.

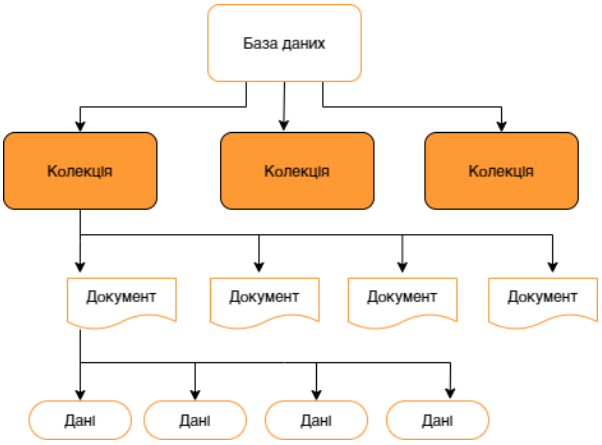


Рисунок 1.6 — Блок схема документно-орієнтована модель даних (виконано самостійно)

1.2.7 Ключ-значення модель даних (Key-Value Model)[22] - це тип бази даних, в якій дані зберігаються у вигляді пар ключ-значення. У цій моделі кожен запис асоціюється з унікальним ключем, який потім використовується для доступу до відповідного значення.

До переваг можна віднести перше це швидкий доступ до даних, оскільки дані зберігаються у вигляді ключів та відповідних значень, пошук та доступ до даних може бути дуже швидким, особливо при використанні оптимізованих структур даних, таких як хеш-таблиці. Наступним є простота в реалізації та використанні адже ключ-значення модель даних є дуже простою та легкою для реалізації та використання, що робить її популярною для швидких прототипів та простих додатків. Ідеально[9] підходить для кешування адже модель є ідеальною для зберігання кешованих даних, оскільки швидкий доступ за ключем дозволяє ефективно зберігати та використовувати кешовані значення. Масштабованість, багато систем ключ-значення можуть бути легко масштабовані горизонтально шляхом розділення даних між різними вузлами.

До недоліків можна віднести обмеженість виразності даних Ключ-значення модель даних є дуже простою, але це також означає, що вона може бути обмеженою виразністю для деяких типів даних або структур. Складнощі у вираженні складних відносин, виразність моделі може страждати від спроб виразити складні відносини між даними, які не можуть бути ефективно представлені у вигляді ключ-значення. Неефективність для деяких типів запитів адже деякі типи запитів можуть бути менш ефективними[7] або навіть неможливими в реалізації за допомогою ключ-значення моделі даних, зокрема, запити, які вимагають складних агрегацій або фільтрацій. Присутня втрата структури даних, у деяких випадках важко зберігати структуру даних в моделі ключ-значення[14], особливо якщо вона потребує вкладених або ієрархічних структур.

Ключ	Значення
user_id	33gh-312kg-4rgm
вік	30
e-mail	anton.shtelma@nure.ua
дата регестрації	2022-08-01

Рисунок 1.7 — Блок схема ключ-значення модель даних (виконано самостійно)

Можна зробити висновок, що кожна з перерахованих моделей баз даних має свої переваги та недоліки, і вибір конкретної моделі залежить від конкретних потреб та характеристик проекту. Залежно від конкретного проекту може варіюватися оптимальність та підходи до кожної моделі бази даних[22]. Тому важливо враховувати специфіку вимог та потреб проекту при виборі найбільш відповідної моделі.

1.3 Аналіз аналогів архітектури баз даних

Наступим етапом було дослідження архітектурних рішень баз даних. Архітектурні рішення баз даних визначають структуру та організацію даних та включають в себе вибір конкретних технологій, методів і підходів для проектування, реалізації та обслуговування баз даних. Найпоширеніші архітектурні рішення включають:

1.3.1 Однорідна архітектура баз даних[14] - це архітектурне рішення, при якому всі компоненти бази даних, такі як обробники запитів, зберігання даних і інтерфейси користувача, об'єднані в єдину систему на одному рівні.

До переваг можна віднести перше це простоту управління адже однорідна архітектура спрощує управління базою даних, оскільки всі компоненти працюють на одному рівні, що робить процес управління більш прозорим і простим.

Наступним є легка масштабованість, завдяки однорідності архітектури, масштабування бази даних може бути більш простим, оскільки можна додавати нові сервери або ресурси, не розриваючи цілісність системи. Присутня оптимізована процесорна та мережева взаємодія тому, що всі компоненти бази даних знаходяться на одному рівні, що дозволяє оптимізувати процесорні та мережеві ресурси для кращої продуктивності та швидкодії. Зменшення загальних витрат на обслуговування завдяки простоті структури та зменшенню складності управління, однорідна архітектура може допомогти знизити витрати на обслуговування та підтримку системи баз даних[10].

До недоліків слід віднести це обмежена масштабованість, адже однорідна архітектура може стати обмеженням при великому масштабуванні, оскільки вона може досягти межі ємності або продуктивності при додаванні нових ресурсів. Одинична точка відмови, адже якщо в системі є одинична точка відмови, наприклад, центральний сервер, він може стати слабким місцем системи та призвести до збоїв або перебоїв у роботі. Також слід віднести і складність розподіленої обробки, при великому масштабі системи можуть виникнути проблеми з розподіленою обробкою даних через обмежену спроможність центрального сервера або мережі[17]. Неоптимальне використання ресурсів, в окремих випадках деякі ресурси можуть бути неоптимальна використані через обмежену можливість розподілу завдань між компонентами системи.

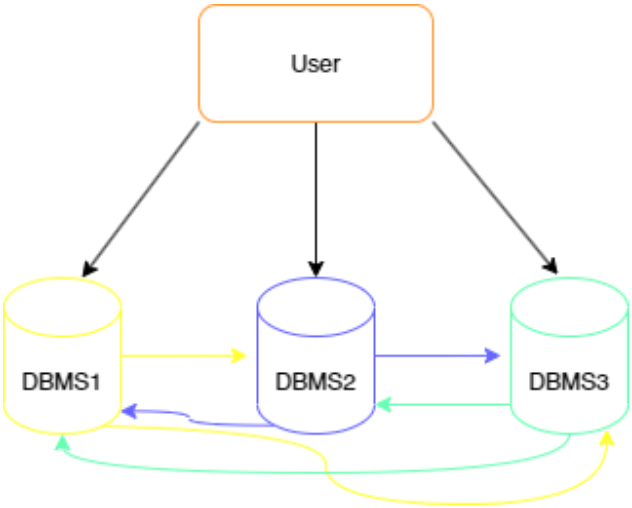


Рисунок 1.8 — Блок схема однорідна архітектура баз даних (виконано самостійно)

1.3.2 Багаторівнева архітектура баз даних - це архітектурне рішення, яке включає декілька рівнів абстракції та функціональності для організації та управління даними[1]. Зазвичай ця архітектура включає рівні доступу до даних, логіки бізнес-логіки та інтерфейсу користувача. Ось деякі переваги та недоліки багаторівневої архітектури баз даних:

До переваг можна віднести розділення обов'язків, багаторівнева архітектура дозволяє чітко розділити функціональні обов'язки між різними рівнями системи, що спрощує розробку, підтримку та масштабування системи. Підвищену безпеку у порівнянні з іншими архітекторами, завдяки розділенню функціональності на різні рівні, багаторівнева архітектура дозволяє регулювати доступ до даних та забезпечувати безпеку на різних рівнях системи. Легше керування даними, ця архітектура сприяє кращому керуванню даними, оскільки дозволяє централізовано управляти базами даних та їхніми взаємозв'язками. Масштабованість, завдяки розділенню функціональності на різні рівні, багаторівнева архітектура дозволяє легко масштабувати систему шляхом додавання або зміни окремих компонентів.

До недоліки можна віднести перше це складність розробки і підтримки, багаторівнева архітектура може бути складною для розробки та підтримки через необхідність управління багатьма рівнями та взаємодією між ними. Потенційні проблеми з продуктивністю, збільшення кількості рівнів може призвести до додаткового завантаження системи та зменшення продуктивності, якщо не враховувати оптимізацію та використання ресурсів. Велика кількість зв'язків між рівнями може зробити систему складною для розуміння та підтримки, особливо великим проектам[1]. Ризик одиничної точки відмови залежність від певних рівнів може створити ризик одиничної точки відмови, коли відмова одного рівня може призвести до збою у всій системі.

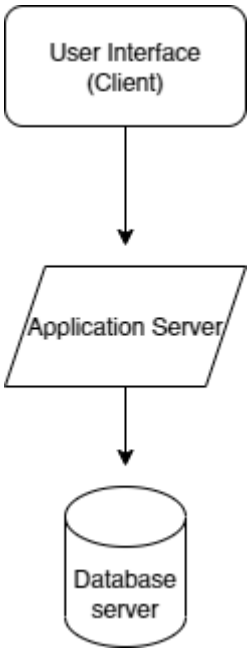


Рисунок 1.9 — Блок схема багаторівнева архітектура баз даних (виконано самостійно)

1.3.3 Розподілена архітектура баз даних (Distributed Database Architecture) - це архітектурне рішення, при якому дані фізично розташовані на різних вузлах комп'ютерної мережі, але з точки зору користувача ці дані здаються логічно об'єднаними та єдиними. До фізично розташованих даних, розміщені на різних фізичних серверах або вузлах мережі, що дозволяє розділити навантаження та забезпечити резервні копії даних. Логічна єдність, незважаючи на розподілений характер, користувачам надається враження єдності бази даних, і вони можуть звертатися до даних як до єдиного цілого. Розподілена обробка запитів можуть бути оброблені паралельно на різних вузлах, що забезпечує покращену продуктивність та швидкодію системи[14].

До переваг слід віднести це високу доступність, адже завдяки реплікації даних та резервним копіям, розподілені бази даних забезпечують високу доступність даних навіть у разі відмови одного з вузлів. Масштабованість коли розподілена архітектура дозволяє легко масштабувати систему, додавши або видаливши вузли в мережі, що дозволяє забезпечити високу продуктивність та працездатність навіть у великих обсягах даних[1]. Географічна розподіленість,

тому що дані можуть бути розподілені по різних регіонах або країнах, що дозволяє забезпечити швидкий доступ до даних користувачам у різних частинах світу.

До недоліків можна віднести складність управління, адже управління розподіленою базою даних може бути складним завданням, оскільки необхідно керувати реплікацією, розподілом даних та забезпеченням їх консистентності. Синхронізація даних забезпечення консистентності даних між різними вузлами може бути складним завданням, особливо в умовах великої кількості транзакцій та паралельних операцій. Витрати на мережеве з'єднання відбувається за рахунок передача даних між різними вузлами може призвести до збільшення витрат на мережеве з'єднання та пропускну спроможність мережі.

Можна зробити короткий висновок, що розподілена архітектура баз даних є потужним інструментом для розробки великих та масштабованих систем, але вона також потребує уважного проектування та управління для забезпечення ефективності та надійності системи.

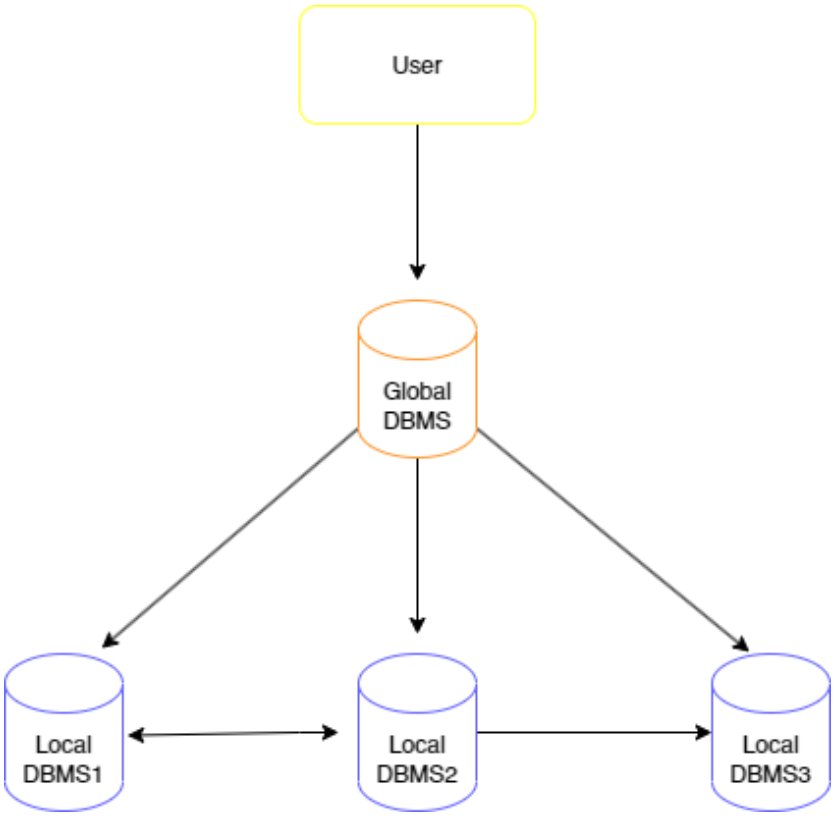


Рисунок 1.10 — Блок схема розподілена архітектура баз даних (виконано самостійно)

1.3.4 Шардинг (Sharding) - це метод розподілу та реплікації даних, який використовується в розподілених системах баз даних для підвищення масштабованості та продуктивності[10]. У шардинговій архітектурі дані розбиваються на невеликі частини, які називаються шардами, і кожен шард зберігається та обробляється окремо. У основні характеристики, переваги та недоліки шардингової архітектури входять розподіл даних це коли дані розбиваються на окремі шарди згідно з певним алгоритмом розподілу, таким як хешування або розподіл за діапазоном значень. Реплікація це коли кожен шард може бути реплікований на кілька копій для забезпечення високої доступності та надійності. Паралельна обробка відбувається завдяки якому шардинг дозволяє розділити навантаження запитів між різними шардами, що дозволяє обробляти запити паралельно та підвищує продуктивність системи. Блок схема шардування зображена нижче.

До переваг можна віднести висока масштабованість адже шардинг дозволяє розділити навантаження між різними шардами, що дозволяє масштабувати систему горизонтально та обробляти більше даних. Наступним є покращена продуктивність[14]. Паралельна обробка запитів на різних шардах дозволяє зменшити час відповіді та покращити продуктивність системи. Висока доступність адже реплікація шардів дозволяє забезпечити високу доступність даних навіть у разі відмови окремих вузлів чи шардів.

А до недоліків можна віднести складність конфігурації її налаштування та управління шардингом де може бути складним завданням, особливо в разі розширення системи та збільшення кількості шардів. Наступним є потреба у моніторингу та керуванні, адже шардинг потребує постійного моніторингу та керування для забезпечення балансування навантаження та попередження проблем з розподілом даних. Складність збільшення шардів це коли додавання нових шардів або зміна конфігурації існуючих може бути складним завданням, особливо в умовах роботи вже функціонуючої системи.

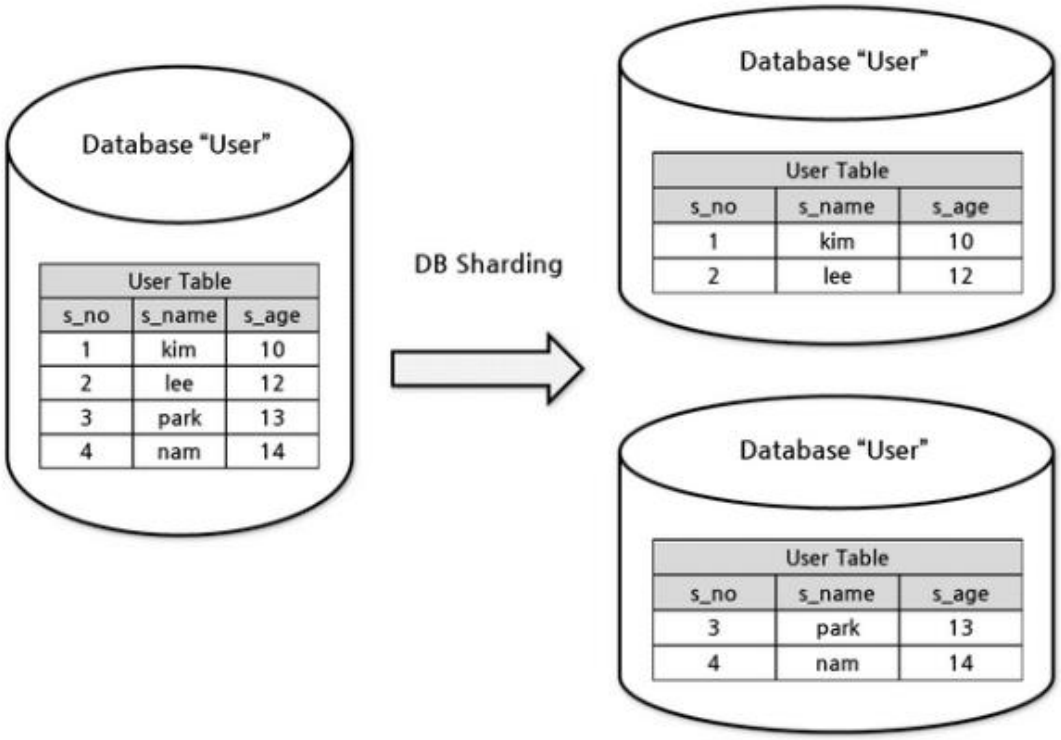


Рисунок 1.11 — Блок схема шардинг (Sharding) [22]

1.3.5 Федеративна архітектура баз даних - це підхід, в якому кілька незалежних баз даних об'єднуються в єдину систему, яка надає єдиний інтерфейс для доступа до даних. Кожна база даних залишається незалежною та може розташовуватися на різних серверах чи навіть в різних мережах. Блок схема зображена нижче

До переваг можна віднести на сам перед це єдина точка доступу Федеративна архітектура дозволяє користувачам отримати доступ до даних з різних джерел через єдиний інтерфейс, що спрощує роботу з даними. Наступним є розподілена обробка, де запити до даних можуть бути розподілені між різними базами даних, що дозволяє покращити продуктивність та забезпечити більш ефективне використання ресурсів. Також присутня гнучкість та масштабованість, де федеративна архітектура дозволяє додавати або видаляти джерела даних без значних змін у програмному забезпеченні, що забезпечує гнучкість та масштабованість системи. Наступною перевагою є збереження існуючих інвестицій, існуючі системи баз даних можуть бути легко інтегровані в

федеративну архітектуру, що дозволяє зберігати існуючі інвестиції в обладнання та програмне забезпечення.

До недоліків такої архітектури можна віднести це складність інтеграції у різних джерел даних у єдину систему може бути складною задачею, особливо якщо вони використовують різні стандарти та протоколи. Також присутня проблема з безпекою, адже забезпечення безпеки даних у федеративній архітектурі може бути складним завданням через розподілених та різнорідних джерел даних[10]. Навантаження на мережу теж можна віднести до мінусів, адже передача даних між різними джерелами може створювати додаткове навантаження на мережу, особливо в умовах великої кількості запитів та обсягів даних. Синхронізацію даних теж можна віднести до недоліків, адже забезпечення консистентності даних між різними джерелами може бути складним завданням, особливо в умовах різних операцій та змін даних.

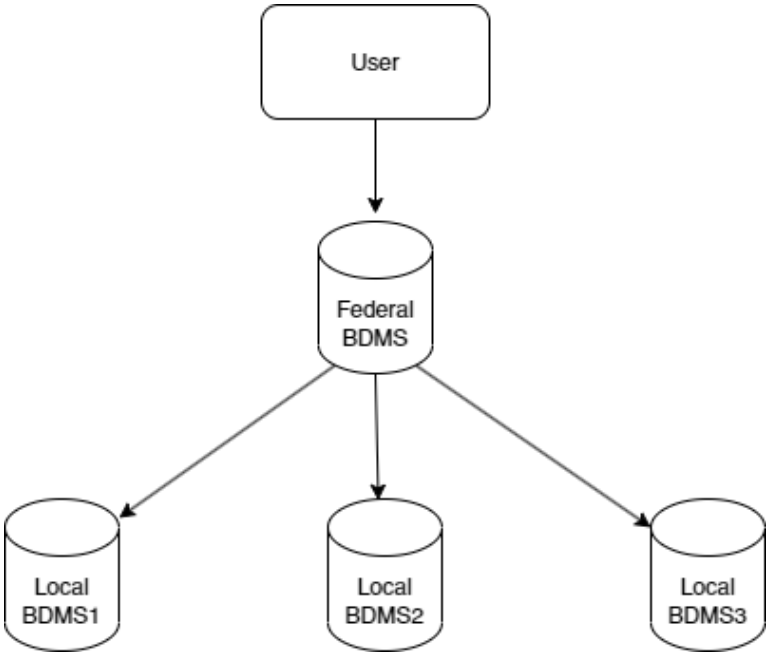


Рисунок 1.12 — Блок схема федеративна архітектура баз даних (виконано самостійно)

Отже проаналізувавши методи та архітектурні рішення можна зробити висновок, що виходячи з потреб треба підбирати необхідний метод та архітектуру, адже у кожного методу та архітектури є свої особливості.

1.4 Постановка задачі

Для дипломної роботи необхідно розробити та впровадити оптимальну модель і архітектуру бази даних, яка відповідатиме потребам організації у зберіганні, обробці та доступі до даних.

Зробити вибір моделі бази даних, а саме розглянути різні моделі баз даних (реляційні, документно-орієнтовані, графові, колонкові, Key-Value, гібридні). Оцінити переваги та недоліки кожної моделі у контексті вимог проекту.

Провезти аналіз та вибір архітектури бази даних. Розглянути різні архітектури (монолітна, розподілена, кластерна, хмарна). Визначити оптимальну архітектуру з урахуванням вимог до продуктивності, масштабованості, відмовостійкості та безпеки.

Також необхідно будет спроектувати базу даних, а саме розробити концептуальну схему бази даних. Створити фізичну модель бази даних, враховуючи вибрану модель і архітектуру. Визначити необхідні індекси, ключі та інші оптимізаційні механізми.

Провести реалізацію та тестування, а саме впровадити базу даних на обраному СУБД. Провести тестування на реальних даних, перевіряючи продуктивність, масштабованість і відмовостійкість системи.

Оптимізація і підтримка,а а саме провести аналіз результатів тестування і внести необхідні корективи. Впровадити механізми моніторингу та резервного копіювання. Розробити план підтримки та оновлення бази даних.

Очікувані результатами буде оптимально вибрана модель і архітектура бази даних, яка забезпечить ефективне зберігання, обробку і доступ до даних. Висока продуктивність системи при роботі з великими обсягами даних. Масштабованість бази даних відповідно до зростання вимог бізнесу. Відмовостійкість та надійність системи. Забезпечення безпеки та конфіденційності даних.

Будуть такі виклики та ризики, а саме складність у виборі оптимальної моделі та архітектури через різноманітність доступних технологій. Потенційні проблеми з продуктивністю та масштабованістю при неправильному виборі моделі або архітектури. Високі вимоги до безпеки даних, особливо у контексті відповідності нормативним вимогам. Необхідність постійної підтримки та оновлення бази даних для забезпечення її ефективної роботи.

Правильний вибір моделі та архітектури бази даних є ключовим фактором успіху будь-якого проекту, що працює з великими обсягами даних. Дана постановка задачі описує підхід до аналізу, вибору, проектування, реалізації та підтримки бази даних, що забезпечить ефективне зберігання та обробку даних відповідно до вимог бізнесу.

2 ВИБІР ОПТИМАЛЬНОГО АЛОРИТМУ КЛАСТЕРИЗАЦІЇ

2.1 Види кластеризаторів баз даних

Кластеризація баз даних полягає в об'єднанні кількох серверів або вузлів у кластер для забезпечення балансування навантаження, підвищення відмовостійкості та масштабованості системи.

До основних аспектів кластеризатора належить балансування навантаження, відмовостійкість, масштабованість.

У балансуванні навантажень кластеризація дозволяє розподіляти запити та обробку даних між кількома серверами, що забезпечує рівномірне використання ресурсів і запобігає перевантаженню окремих вузлів. Як приклад можна навести у системі з високим трафіком (наприклад, інтернет-магазин) кластер серверів може обробляти запити користувачів одночасно, знижуючи затримки та покращуючи продуктивність.

Якщо казати про відмовостійкість то якщо один сервер у кластері виходить з ладу, інші сервери продовжують працювати, забезпечуючи безперервність роботи системи. Це знижує ризик втрати даних і підвищує надійність. Можна навести приклад, в системах онлайн-банкінгу, де відмовостійкість є критично важливою, кластеризація забезпечує безперервний доступ до даних і послуг навіть у випадку збоїв окремих серверів.

У масштабованості додавання нових серверів до кластера дозволяє легко збільшити потужність та продуктивність системи у відповідь на зростаючі вимоги. Як приклад соціальні мережі, можуть додавати нові сервери до своїх кластерів, щоб обробляти зростаючі обсяги даних та запити користувачів.

Типи кластеризації бувають бувають активно-активний та активно пасивні. До активно-активний кластер відносять коли усі сервери в кластері працюють одночасно, обробляючи запити і забезпечуючи високу продуктивність. Як приклад можна навести, у великих веб-додатках, таких як пошукові системи, активно-

активні кластери можуть обробляти запити з різних регіонів світу одночасно, забезпечуючи швидкий і надійний доступ.

А як активно-пасивний кластер це коли один сервер працює в активному режимі, тоді як інші залишаються в режимі очікування та стають активними лише у випадку збою основного сервера. Приклад: у банківських системах активно-пасивний кластер може використовуватися для забезпечення безперервності операційних процесів у разі збоїв.

Як найпоширеніших прикладах використання кластеризації у реляційних або NoSQL базах даних. Реляційні бази даних такі як PostgreSQL[21] з використанням Patroni для забезпечення високої доступності, MySQL[21] з використанням Galera Cluster або NoSQL бази даних такі як Apache Cassandra, MongoDB з реплікацією та шардингом.

2.2 Найпоширеніші види кластеризаторів

2.2.1 DBSCAN (Density-Based Spatial Clustering of Applications with Noise)[7]

- це метод кластеризації, який базується на щільності точок в просторі. Основна ідея полягає в тому, що кластери є областями з високою щільністю точок, розділеними областями з низькою щільністю (або шумом). Алгоритм має два основних параметри, Eps (ε) - радіус, в межах якого шукаються сусіди для даної точки та MinPts - мінімальна кількість точок, необхідна для утворення кластера.

Точки діляться на три категорії на основні точки (Core Points) де точки, які мають не менше ніж MinPts сусідів у радіусі Eps і прикордонні точки (Border Points) де точки, які мають менше ніж MinPts сусідів, але знаходяться в межах Eps від основної точки, та шум (Noise) де точки, які не є ані основними, ані прикордонними.

Принцип алгоритму полягає у пошук сусідів тобто для кожної точки знаходиться список сусідів у радіусі Eps. Наступним етапом є класифікація точок це коли кожна точка класифікується як основна, прикордонна або шум.

Формування кластерів відбувається по наступному. Початок з будь-якої точки, яка не була ще відвідана. Якщо точка є основною, створюється новий кластер і всі її сусіди (в межах Eps) додаються до цього кластера. Для кожного нового сусіда (основної точки) повторюється процес, доки не будуть знайдені всі точки, що належать до кластера.

Наступним етапом є повторення, тобто процес повторюється для кожної точки, поки всі точки не будуть класифіковані.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN

# Генерація випадкових даних
np.random.seed(0)
X = np.random.randn(300, 2)

# Параметри DBSCAN
eps = 0.3
min_samples = 5

# Застосування DBSCAN
db = DBSCAN(eps=eps, min_samples=min_samples).fit(X)
labels = db.labels_

# Візуалізація результатів
unique_labels = set(labels)
colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1, len(unique_labels))]

for k, col in zip(unique_labels, colors):
    if k == -1:
        # Шум
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = X[class_member_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)

plt.title('DBSCAN: приклад кластеризації')
plt.show()
```

Рисунок 2.1 — Density-Based Spatial Clustering of Applications with Noise (виконано самостійно)

Принцип алгоритму котрий зображений на рисунку 1 наступний. Перше це генерація даних, ми створюємо набір випадкових двовимірних точок. Наступним етапом є параметри DBSCAN, котрі встановлюємо `eps` та `min_samples` для визначення сусідів та мінімальної кількості точок у кластері. Застосування DBSCAN ми виконуємо кластеризацію, визначаючи мітки для кожної точки. Відображаємо результати, де кожен кластер має свій колір, а точки шуму відмічені чорним, щоб було видно наглядніше. Мова написання коду Python.

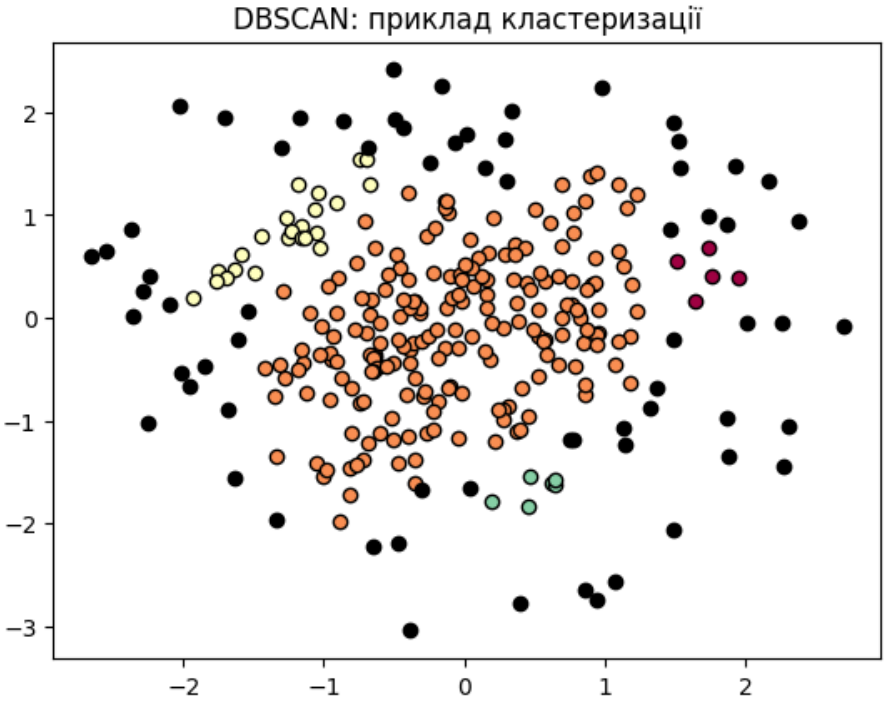


Рисунок 2.2 — Результат алгоритму DBSCAN (виконано самостійно)

2.2.2 Ієрархічна агломеративна кластеризація (HAC) - це метод кластеризації, який створює ієрархію кластерів шляхом поступового об'єднання або розділення кластерів. Агломеративний підхід починається з кожної точки як окремого кластера і поступово об'єднує найближчі кластери, поки всі точки не опиняться в одному кластері або не буде досягнуто певної кількості кластерів.

Основні кроки алгоритму полягають у тому що кожна точка є окремим кластером. Вимірювання відстаней між усіма парами кластерів. Відбувається об'єднання двох найближчих кластерів.

Після об'єднання, матриця відстаней оновлюється для відображення відстані між новим кластером і всіма іншими кластерами. Виконується повторення кроки 2-4 повторюються, доки не буде досягнуто бажаної кількості кластерів або всі точки не опиняться в одному кластері. Методи вимірювання відстаней між кластерами відбуваються методом Single Linkage (найближчий сусід): Відстань між двома кластерами визначається як мінімальна відстань між будь-якими двома точками з різних кластерів. Complete Linkage (найдальніший сусід) відстань між двома кластерами визначається як максимальна відстань між будь-якими двома точками з різних кластерів. Average Linkage (середня відстань) відстань між двома кластерами визначається як середня відстань між усіма парами точок з різних кластерів. Ward's Method відстань між двома кластерами визначається як збільшення суми квадратів помилок після об'єднання кластерів.

Розглянемо нижче приклад кластеризатора ієрархічна агломератна кластеризація. Мова написання кластеризатора Python з використанням додаткових бібліотек[2].

Роз’яснення стосовно кластерного коду нижче, генерація даних. Ми створюємо набір випадкових двовимірних точок з чотирма центрами. Ми застосовуємо агломеративну кластеризацію, використовуючи метод Уорда для визначення відстаней між кластерами.

Метод Уорда, або Ward's method, є одним із методів агломеративної ієрархічної кластеризації. Він мінімізує загальну суму квадратів відхилень точок від їхніх центрів, що робить його схожим на методи аналізу дисперсії. Метод Уорда працює шляхом обчислення відстані між кластерами на основі зміни суми квадратів відхилень, що виникає при їх об'єднанні. Формально, відстань між двома кластерами A і B визначається як:

$$d(A,B) = \frac{|A|\times|B|}{|A|+|B|} \times \|c_A - c_B\|^2 \qquad (2.1)$$

де

|A|і|B| – кількість точток у кластерах A і B відповідно,

$c_A - c_B$ - центроїди (центри мас) кластерів A і B,

$\|c_A - c_B\|^2$- квадрат евклідової відстані між центроїдами кластерів A і B.

тобто метод Уорда прагне мінімізувати збільшення загальної дисперсії при об'єднанні двох кластерів[8]. Іншими словами, він вибирає для об'єднання такі два кластери, які призведуть до найменшого збільшення внутрішньо групової суми квадратів.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering

# Генерація випадкових даних
np.random.seed(0)
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# Виконання агломеративної кластеризації
agg_clustering = AgglomerativeClustering(n_clusters=4, affinity='euclidean', linkage='ward')
labels = agg_clustering.fit_predict(X)

# Візуалізація результатів
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='rainbow', s=30)
plt.title('Hierarchical Agglomerative Clustering')
plt.show()

# Створення дендрограми
linked = linkage(X, 'ward')
plt.figure(figsize=(10, 7))
dendrogram(linked, orientation='top', distance_sort='descending', show_leaf_counts=True)
plt.title('Dendrogram')
plt.show()
```

Рисунок 2.3 — Hierarchical Agglomerative Clustering (виконано самостійно)

На відображенні результату кластеризації видно, що кожен кластер має свій колір.

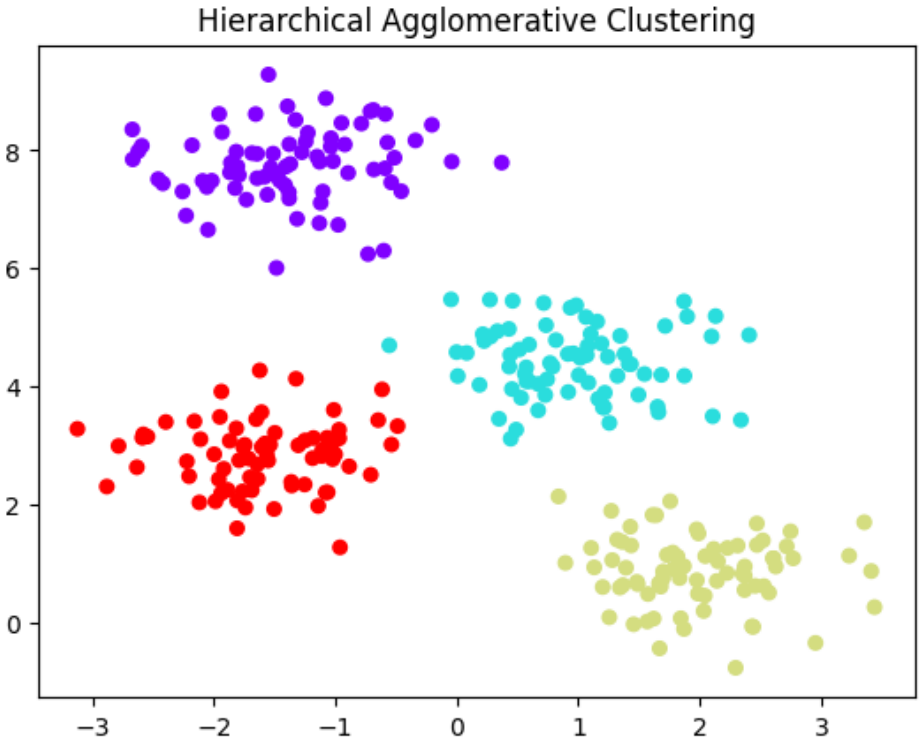


Рисунок 2.4 — Ієрархічна агломеративна кластеризація результат (виконано самостійно)

А при створенні дендрограми вона відображає ієрархічну структуру кластерів.

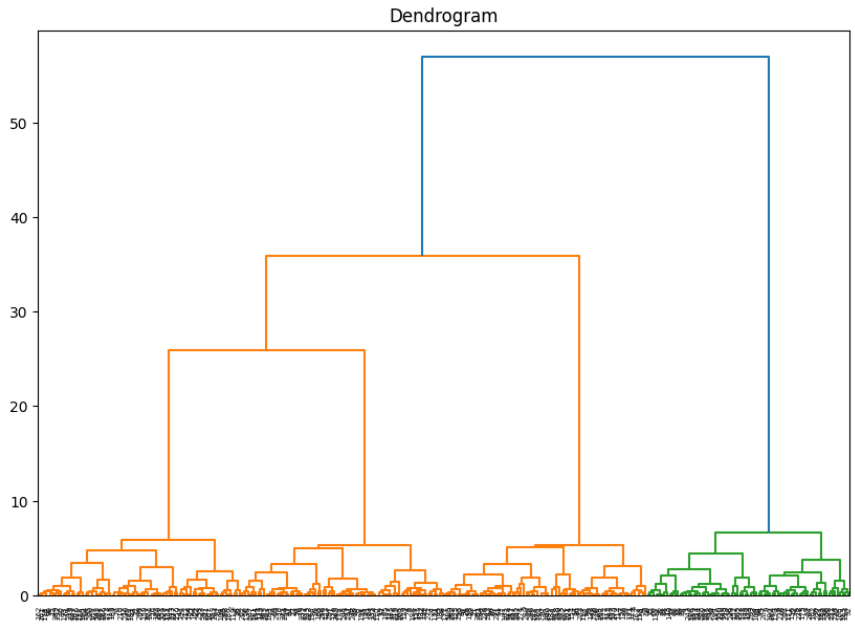


Рисунок 2.5 — Дендрограма ієрархічної кластеризації (виконано самостійно)

Цей алгоритм завершується, коли всі точки об'єднані в один кластер або досягнуто бажаної кількості кластерів. Дендрограма дає змогу візуально оцінити, як кластери утворюються і об'єднуються на різних рівнях ієрархії, що допомагає у виборі оптимальної кількості кластерів.

2.2.3 OPTICS - це метод кластеризації, схожий на DBSCAN, але з деякими вдосконаленнями. Його основна мета - виявити основні структури кластерів у даних, включаючи кластери з різною щільністю. На відміну від DBSCAN, OPTICS не створює безпосередньо набір кластерів. Натомість він створює впорядковану репрезентацію точок, яка полегшує визначення кластерів з різною щільністю.

До основних понять OPTICS входять core distance (ядрова відстань): мінімальна відстань, необхідна для того, щоб точка стала ядровою точкою, тобто вона повинна мати щонайменше minPts сусідів у цій відстані. Reachability Distance (досяжність) це відстань, яка використовується для вимірювання щільності навколо точки в контексті інших точок. Всі точки позначаються як необроблені[16]. Пошук сусідів для кожної необробленої точки обчислюється ядрова відстань і досяжність для всіх сусідів. Порядок обробки точок відбувається у порядку, який мінімізує досяжність, створюючи впорядковане представлення. Побудова кластерів відбувається на основі отриманого порядку точок завдяки якому можна визначити кластери, використовуючи різні пороги щільності.

Розглянемо приклад з двовимірними даними, де ми застосуємо кластеризатор.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import OPTICS, cluster_optics_dbscan
from sklearn.datasets import make_blobs

# Генерація випадкових даних
np.random.seed(0)
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# Виконання OPTICS кластеризації
optics_clustering = OPTICS(min_samples=5, xi=0.05, min_cluster_size=0.1)
optics_clustering.fit(X)

labels = optics_clustering.labels_

# Візуалізація результатів кластеризації
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='rainbow', s=30)
plt.title('OPTICS Clustering')
plt.show()

# Візуалізація досяжності
reachability = optics_clustering.reachability_[optics_clustering.ordering_]
plt.plot(reachability)
plt.title('Reachability Plot')
plt.xlabel('Sample index')
plt.ylabel('Reachability distance')
plt.show()
```

Рисунок 2.6 — Алгоритм кластеризатора OPTICS (виконано самостійно)

Спершу робимо генерація даних створюємо набір випадкових двовимірних точок з чотирма центрами. Використовуємо клас OPTICS зі sklearn для виконання кластеризації. Вказуємо параметри min_samples, xi, min_cluster_size. Приклад виконаний на мові Python. Наступним етапом отримуємо результат

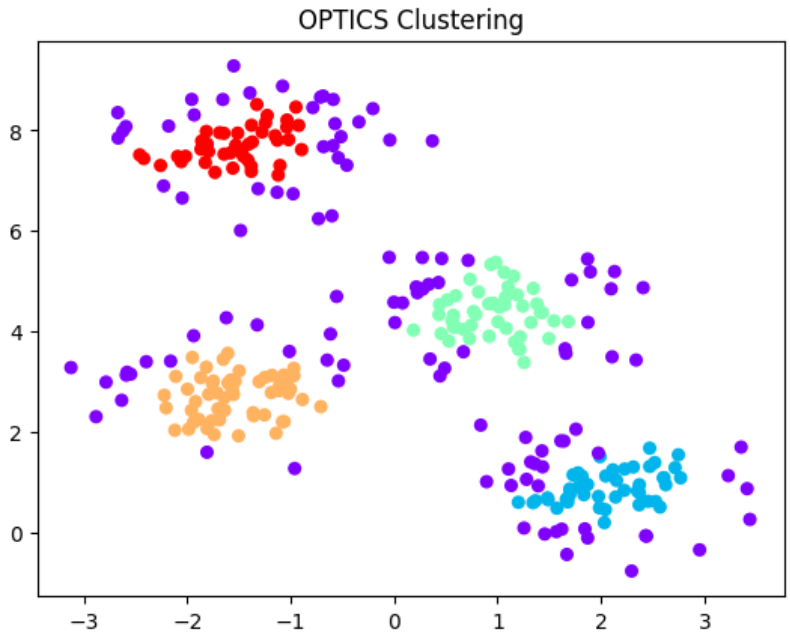


Рисунок 2.7 — Візуальний результат кластеризатора OPTICS(виконано самостійно)

З отриманого результату кластеризації бачимо, що кожен кластер має свій колір.

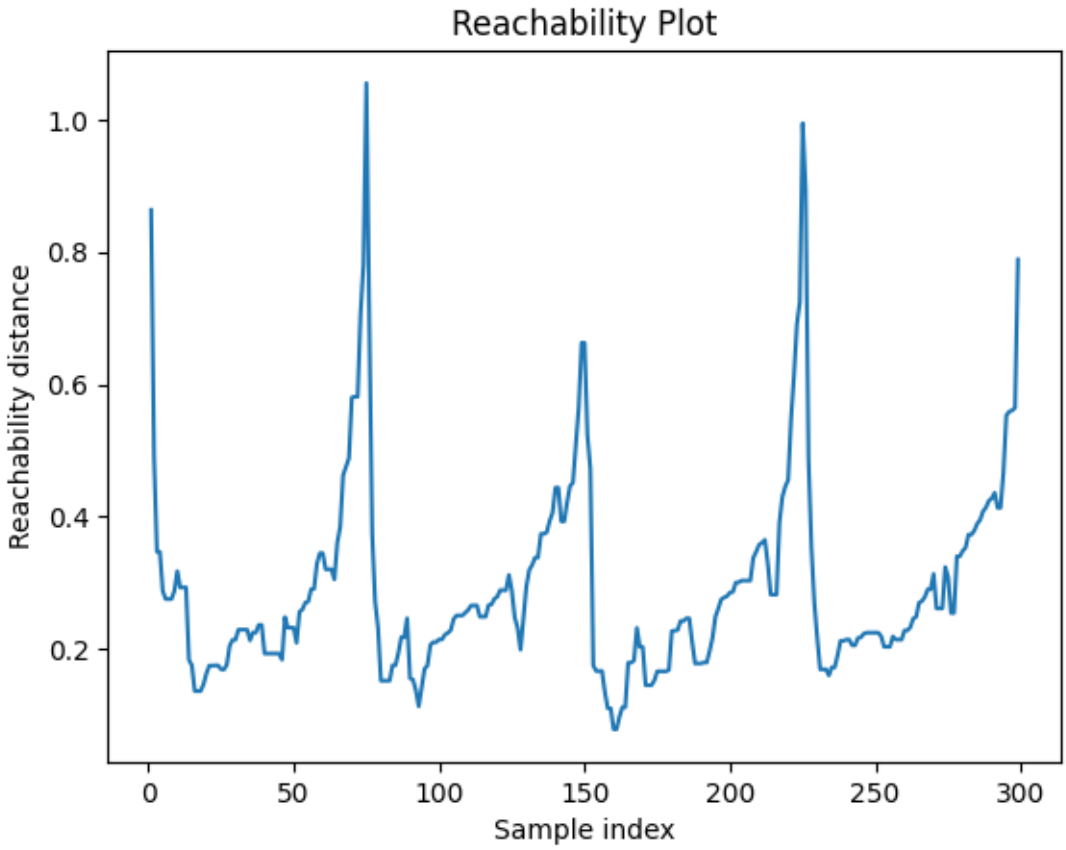


Рисунок 2.8 — Графік досяжності кластеризатора OPTICS (виконано самостійно)

Графік досяжності, який відображає щільність точок у впорядкованому представленні.

Отже можна зробити наступний висновок, що OPTICS виявляє кластери з різною щільністю, дозволяючи візуалізувати структури даних і визначати кластери, які могли б бути пропущені іншими методами кластеризації. Reachability Plot допомагає зрозуміти структуру кластерів і обрати пороги для побудови кластерів

2.2.4 K-Means - це один з найпопулярніших алгоритмів кластеризації, який використовується для розділення набору даних на “k” кластерів, де “k” визначається заздалегідь. Мета алгоритму полягає в мінімізації відстані між точками даних і центроїдами їхніх кластерів[16].

Отже алгоритм працює наступним чином, потрібно обрати “k” початкових центроїдів випадковим чином з набору даних. Призначити кожну точку до найближчого центроїда. Оновлення центроїдів відбувається за рахунок обчислення новий центроїд кожного кластера як середнє значення точок, що належать до цього кластера. Повторення кроків 2 і 3, доки центроїди не перестануть змінюватись або не буде досягнуто максимальну кількість ітерацій. Алгоритм сходиться до розв'язку, де всі точки даних віднесені до кластерів таким чином, що сумарна внутрішньо кластерна варіація мінімізована.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# Генерація випадкових даних
np.random.seed(0)
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# Виконання K-Means кластеризації
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)
labels = kmeans.predict(X)
centroids = kmeans.cluster_centers_

# Візуалізація результатів кластеризації
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='rainbow', s=30)
plt.scatter(centroids[:, 0], centroids[:, 1], s=300, c='black', marker='X')
plt.title('K-Means Clustering')
plt.show()
```

Рисунок 2.9 — Алгоритм кластеризатора K-Means (виконано самостійно)

На рисунку 2.9 приклад з двовимірними даними, де ми застосуємо K-Means кластеризацію. Мова програмування Python.

Створюємо набір випадкових двовимірних точок з чотирма центрами. Використовуємо клас KMeans зі sklearn для виконання кластеризації. Вказуємо параметр n_clusters=4 для визначення кількості кластерів.

Рисунок 2.10 — Візуалізація кластеризатора K-Means (виконано самостійно)

Результати кластеризації можна побачити вижче на рисунку 2.10, де кожен кластер має свій колір, а центроїди позначені чорними хрестиками.

До переваг алгоритму можна віднести простоту і швидкість. Алгоритм K-Means простий у реалізації і швидкий в обчисленнях. Добре масштабується на великі набори даних.

До недоліків слід віднести фіксована кількість кластерів, адже потрібно заздалегідь знати кількість кластерів “k”. Також результат залежить від початкових значень центроїдів. Працює найкраще для сферичних кластерів з однаковою дисперсією.

Зробивши підсумок можна сказати, що K-Means - це ефективний і популярний метод кластеризації, який добре підходить для багатьох задач, де форма і кількість кластерів чітко визначені. У дипломному проекті саме він і буде використовуватися.

2.2.5 Gaussian Mixture Model (GMM) - це імовірнісний модель кластеризації, яка представляє дані як суміш кількох гаусіанських (нормальних) розподілів. Кожен кластер описується своїм гаусіанським розподілом з певними параметрами (середнє значення і коваріаційна матриця). На відміну від K-Means, GMM дозволяє кластерам мати еліптичну форму, а не сферичну[16].

Основні етапи алгоритму GMM складаються з вибору початкових параметрів для кожного гаусіанського розподілу, таких як середнє, коваріаційна матриця і ваги кластерів. Обчислення ймовірностей “E-step”належності кожної точки до кожного кластеру на основі поточних параметрів (це називається "відповідальністю"). Оновлення параметрів гаусіанських розподілів на основі обчислених ймовірностей для максимізації “M-step”. Повторення кроків E і M до збіжності (тобто поки зміни в параметрах не стануть незначними).

Ваги кластерів π_k це частка точок, що належать до кластера (k). Розраховується за наступною формулою[8]:

$$\pi_k = \frac{N_k}{N} \tag{2.2}$$

де

π_k- вага кластера k,

N_k – кількість точок, віднесених до кластера k,

N – загальна кількість точок у наборі даних

Кількість точок N_k віднесених до кластера k, обчислюється як сума відповідальностей $(\gamma(z_i k))$ для всіх точок даних де $\gamma(z_i k)$ – ймовірність того, що точка і належить кластеру k[8]:

$$N_k = \sum_{i=1}^{N} \gamma(z_i k) \quad (2.3)$$

$\gamma(z_i k)$ – відповідальність для точки іта кластера k, обчислюється під час Е-кроку ЕМ-алгоритму:

$$\gamma(z_i k) = \frac{\pi_k N(x_i|\mu_k,\Sigma_k \quad)}{\sum_{j=1}^{K} \pi_j N(x_i|\mu_j,\Sigma_j \quad)} \quad (2.4)$$

Середнє μ_k, центр гаусіанського розподілу для кластера k. Формула для середнього значення наступна:

$$\mu_k = \frac{\sum_{i=1}^{N} \gamma(z_i k) x_i}{N_k} \quad (2.5)$$

де

μ_k – нове середнє значення для кластера k,

$\gamma(z_i k)$ – відповідальність для точки і та кластера k, яка обчислюється під час Е-кроку;

x_i – координати точки і,

$N_k = \sum_{i=1}^{N} \gamma\,(z_i k)$ – сумарна відповідальність для кластера k.

Коваріаційна матриця $\sum_k$, визначає форму гаусіанського розподілу для кластера k. Вона обчислюється під час М-кроку алгоритму максимізації очікування (ЕМ) на основі точок, які належать до цього кластера, з урахуванням відповідальностей ($\gamma(z_i k)$).

$$\sum_k = \frac{\sum_{i=1}^{N} \gamma(z_i k)(x_i - \mu_k)(x_i - \mu_k)^T}{N_k} \quad (2.6)$$

де

$\sum_k$ - нова коваріаційна матриця для кластера k,

$\gamma(z_i k)$ – відповідальність для точки і та кластера k, яка обчислюється під час Е-кроку,

x_i – координати точки і,

μ_k – нове середнє значення для кластера k, обчислюване раніше,

$N_k = \sum_{i=1}^{N} \gamma(z_i k)$ – сумарна відповідальність для кластера k.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.mixture import GaussianMixture

# Генерація випадкових даних
np.random.seed(0)
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# Виконання GMM кластеризації
gmm = GaussianMixture(n_components=4, covariance_type='full')
gmm.fit(X)
labels = gmm.predict(X)

# Візуалізація результатів кластеризації
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='rainbow', s=30)
plt.title('GMM Clustering')
plt.show()
```

Рисунок 2.11 — Алгоритм кластеризатора GMM (виконано самостійно)

Приклад зроблений з двовимірними даними, де ми застосуємо GMM кластеризацію.

Приниципп роботи у наступному, спершу створюємо набір випадкових двовимірних точок з чотирма центрами. Наступним використовуємо клас `GaussianMixture` зі `sklearn` для виконання кластеризації. Вказуємо параметр `n_components=4` для визначення кількості кластерів і `covariance_type='full'` для використання повної коваріаційної матриці.

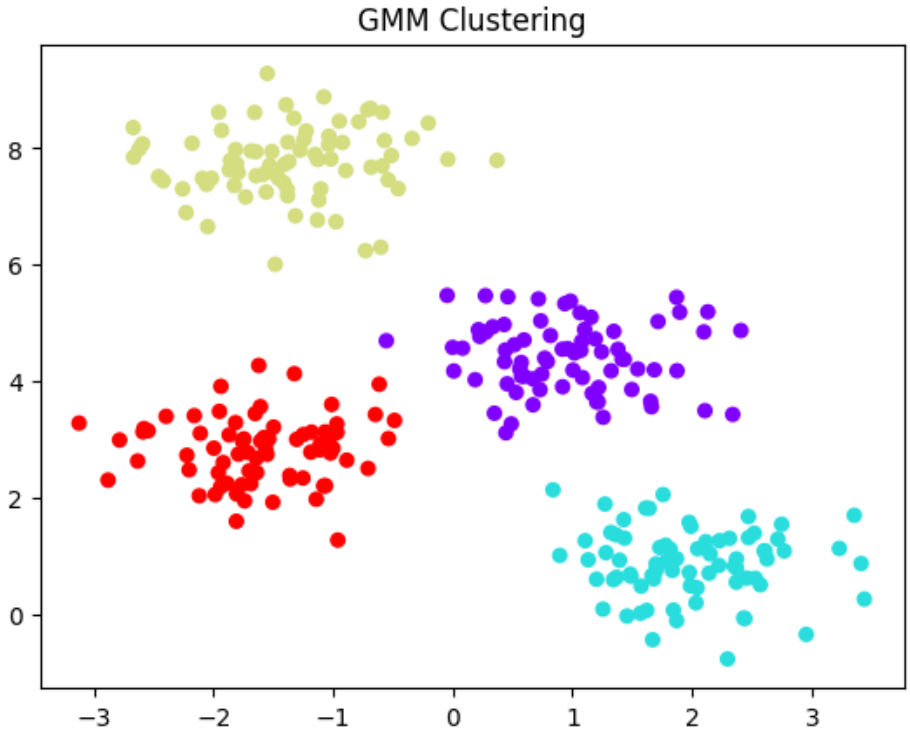


Рисунок 2.12 — Візуалізація кластеризатора GMM (виконано самостійно)

Візуалізація результатів кластеризації, де кожен кластер має свій колір.

Отже переваги кластера GMM це гнучкість, адже завдяки кластеризатору GMM може моделювати кластери різної форми і щільності. Забезпечує ймовірності належності точок до кластерів, що може бути корисним для подальшого аналізу.

До недоліків слід віднести чутливість до ініціалізації, адже результати можуть залежати від початкових умов, хоча існують методи для поліпшення ініціалізації. Наступний його недолік це кількість кластерів, адже необхідно заздалегідь знати кількість кластерів k. Складність обчислювання, GMM може бути обчислювально інтенсивним для великих наборів даних і великої кількості кластерів.

Висновок можна зробити, що GMM - це потужний інструмент для кластеризації, який дозволяє виявляти складні структури в даних і працює добре в багатьох різних контекстах.

Він дозволяє ефективно розподіляти навантаження між серверами, забезпечуючи надійність та безперервний доступ до даних навіть у разі відмов

окремих серверів. Це робить кластеризацію критично важливою для систем, що потребують високої доступності та продуктивності.

Було прийнято рішення обрати для майбутньої реляційної бази даних кластеризатор K-середніх (K-Means) один із найпоширеніших і найпростіших у розумінні методів кластеризації. Він розбиває дані на заздалегідь задану кількість кластерів (k) шляхом мінімізації середньоквадратичної відстані між точками даних та центрами кластерів.

3 ВИБІР ОПТИМАЛЬНОГО АЛОРИТМУ СТИСНЕННЯ

3.1 Алгоритми стиснення

3.1.1 Алгоритм Delta-стиснення принцип роботи полягає в зменшенні розміру даних шляхом збереження різниць (дельт) між послідовними елементами, а не самих елементів. Це ефективно для даних з високою кореляцією між сусідніми значеннями.

Розглянемо алгоритм у дії, тож нехай у нас є послідовність чисел x_1, x_2, ….,x_n. Розрахунок обчислень дельт це різниці між послідовними елементами обчислюються як для $d_i = x_i - x_{i-1}$ для i=2, … n. Перше значення d_i буде дорівнювати x_1. Стиснення дельт відбувається за допомогою будь-якого традиційного методу стиснення (наприклад, методів ентропійного стиснення, таких як кодування Хаффмана(буде продемонстровано нижче) чи LZW).

Стосовно коду котрий буде нижче, функція delta_encode обчислює дельти для вхідних даних. Перше значення зберігається без змін, а для кожного наступного обчислюється різниця з попереднім. Функція delta_decode відновлює оригінальні дані з дельт. Перше значення зберігається без змін, а кожне наступне відновлюється шляхом додавання попереднього значення. Функція стиснення compress_data стискає закодовані дельти за допомогою бібліотеки zlib. Zlib бібліотека — це популярна бібліотека для стиснення та розтискання даних, яка забезпечує високоефективні алгоритми стиснення без втрат. Бібліотека zlib використовується у багатьох програмних продуктах і форматах даних, таких як gzip, PNG та багато інших. Функція decompress_data розтискає дані і декодує їх назад у початкову послідовність. Код продемонстровано на рисунку 12, мова програмування Python.

```python
import zlib
import matplotlib.pyplot as plt

def delta_encode(data):
    """Delta encoding of the input data."""
    if not data:
        return []
    deltas = [data[0]]
    for i in range(1, len(data)):
        deltas.append(data[i] - data[i - 1])
    return deltas

def delta_decode(deltas):
    """Delta decoding of the input deltas."""
    if not deltas:
        return []
    data = [deltas[0]]
    for i in range(1, len(deltas)):
        data.append(data[i - 1] + deltas[i])
    return data

def compress_data(data):
    """Compress the input data using zlib."""
    encoded_data = delta_encode(data)
    compressed_data = zlib.compress(bytes(encoded_data))
    return compressed_data

def decompress_data(compressed_data):
    """Decompress the input data using zlib."""
    decompressed_data = list(zlib.decompress(compressed_data))
    decoded_data = delta_decode(decompressed_data)
    return decoded_data

# Приклад використання
original_data = [100, 102, 105, 107, 110]
print("Original data:", original_data)

# Стиснення даних
compressed_data = compress_data(original_data)
print("Compressed data:", compressed_data)

# Розтиснення даних
decompressed_data = decompress_data(compressed_data)
print("Decompressed data:", decompressed_data)

# Перевірка, чи розтиснуті дані відповідають оригінальним
assert decompressed_data == original_data

# Візуалізація результатів
fig, axs = plt.subplots(2, 1, figsize=(10, 8))

# Оригінальні дані
axs[0].plot(original_data, marker='o', linestyle='-', color='b')
axs[0].set_title('Вихідні дані')
axs[0].set_xlabel('Index')
axs[0].set_ylabel('Value')

# Розтиснуті дані
axs[1].plot(decompressed_data, marker='o', linestyle='-', color='r')
axs[1].set_title('Стистнені дані ')
axs[1].set_xlabel('Index')
axs[1].set_ylabel('Value')

plt.tight_layout()
plt.show()

```

Рисунок 3.1 — Алгоритм коду стиснення Delta (виконано самостійно)

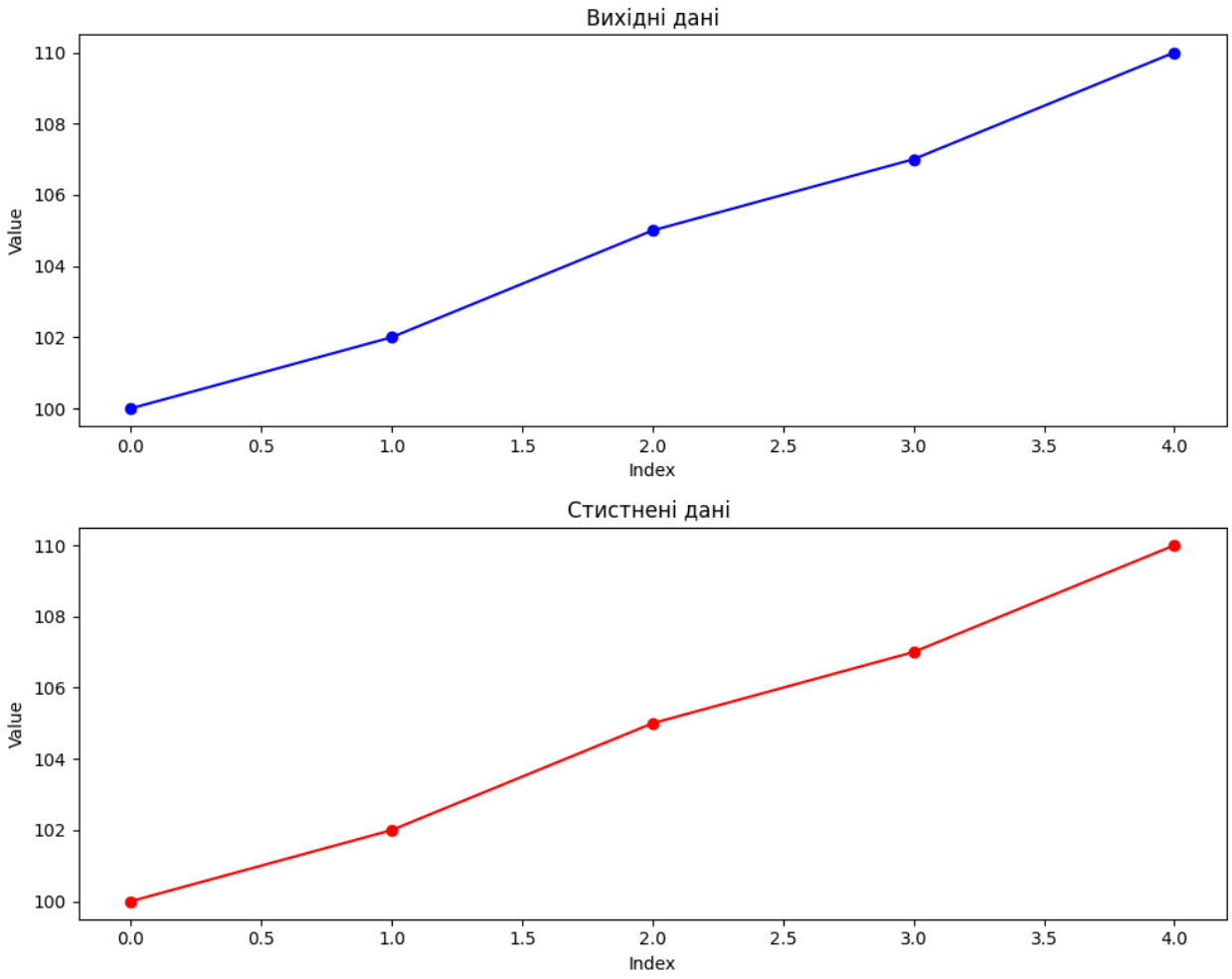


Рисунок 3.2 — Графік результату стиснення алгоритму Delta (виконано самостійно)

Графіки показують, що розтиснуті дані відповідають оригінальним, підтверджуючи коректність роботи алгоритму. Цей приклад демонструє нам як можна ефективно використовувати Delta-стиснення для зменшення розміру даних і зберігання їх у стислому вигляді з можливістю відновлення оригінальних значень.

Цей код стискає послідовність чисел, використовуючи Delta-стиснення, а потім розтискає її, відновлюючи початкові дані. Як видно, розтиснуті дані відповідають оригінальним, що підтверджує правильність роботи алгоритму.

3.1.2 Алгоритм Huffman — це метод стиснення даних без втрат, який базується на частоті появи символів у даних. Основна ідея полягає у використанні коротших кодів для символів, які зустрічаються частіше, і довших кодів для символів, які зустрічаються рідше.

Принцип алгоритму полягає у підрахунку частоти сиволів, побудови пріоритетної черги, подубові дерева Huffman, та генерація кодів Huffman. Отже спершу відбувається підрахунок частоти символів, обчислення частоти кожного символу у вхідних даних, у нашому випадку це речення: “ this is an example for huffman encoding ”. Наступний етапом є побудова пріоритетної черги, отже створення пріоритетної черги (heap) на основі частоти символів. Наступним є побудова дерева Huffman, вона відбувається за рахунок рекурсивного з'єднання найменш частих символів у дереві. Останнім є генерація кодів Huffman, вона відбувається за рахунок присвоєння бітових кодів кожному символу на основі побудованого дерева. Як експеримент було створено код для наочного бачення експерименту. Зображений на рисунку 14. Принцип виконання алгоритму наступний, підрахунок частоти символів: використовується функція “Counter” для обчислення частоти кожного символу у вхідних даних. Застосовується так звана пріоритетна черга “heapq” для побудови дерева Huffman, де вузли з найменшою частотою об'єднуються в нові вузли до тих пір, поки не залишиться один вузол. Генерація кодів Huffman відбувається рекурсивно обходячи дерева для присвоєння бітових кодів кожному символу. Стискаються дані шляхом заміни кожного символу його бітовим кодом та відновлюються оригінальні дані з бітового представлення. Мова виконання коду Phyton.

```python
import heapq
from collections import Counter
import matplotlib.pyplot as plt

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(frequencies):
    heap = [Node(char, freq) for char, freq in frequencies.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        merged = Node(None, node1.freq + node2.freq)
        merged.left = node1
        merged.right = node2
        heapq.heappush(heap, merged)

    return heap[0]

def generate_codes(node, prefix="", codebook={}):
    if node is not None:
        if node.char is not None:
            codebook[node.char] = prefix
        generate_codes(node.left, prefix + "0", codebook)
        generate_codes(node.right, prefix + "1", codebook)
    return codebook

def huffman_encode(data):
    frequencies = Counter(data)
    huffman_tree = build_huffman_tree(frequencies)
    huffman_codes = generate_codes(huffman_tree)

    encoded_data = "".join(huffman_codes[char] for char in data)
    return encoded_data, huffman_codes, frequencies

def huffman_decode(encoded_data, huffman_codes):
    reverse_codes = {v: k for k, v in huffman_codes.items()}
    current_code = ""
    decoded_data = []

    for bit in encoded_data:
        current_code += bit
        if current_code in reverse_codes:
            decoded_data.append(reverse_codes[current_code])
            current_code = ""

    return "".join(decoded_data)

# Приклад використання
data = "this is an example for huffman encoding"
print("Original data:", data)

# Стиснення даних
encoded_data, huffman_codes, frequencies = huffman_encode(data)
print("Encoded data:", encoded_data)

# Розтиснення даних
decoded_data = huffman_decode(encoded_data, huffman_codes)
print("Decoded data:", decoded_data)

# Перевірка, чи розтиснуті дані відповідають оригінальним
assert decoded_data == data

# Візуалізація результатів
fig, axs = plt.subplots(2, 1, figsize=(10, 8))

# Оригінальні дані
axs[0].bar(frequencies.keys(), frequencies.values(), color='b')
axs[0].set_title('Вихідна Data')
axs[0].set_xlabel('Character')
axs[0].set_ylabel('Frequency')

# Розподіл кодів Хаффмана
code_lengths = [len(huffman_codes[char]) for char in frequencies.keys()]
axs[1].bar(frequencies.keys(), code_lengths, color='r')
axs[1].set_title('Huffman алгоритм')
axs[1].set_xlabel('Character')
axs[1].set_ylabel('Code Length')

plt.tight_layout()
plt.show()
```

Рисунок 3.3 — Код алгоритму Huffman (виконано самостійно)

Після виконання коду отримали наступні результати.

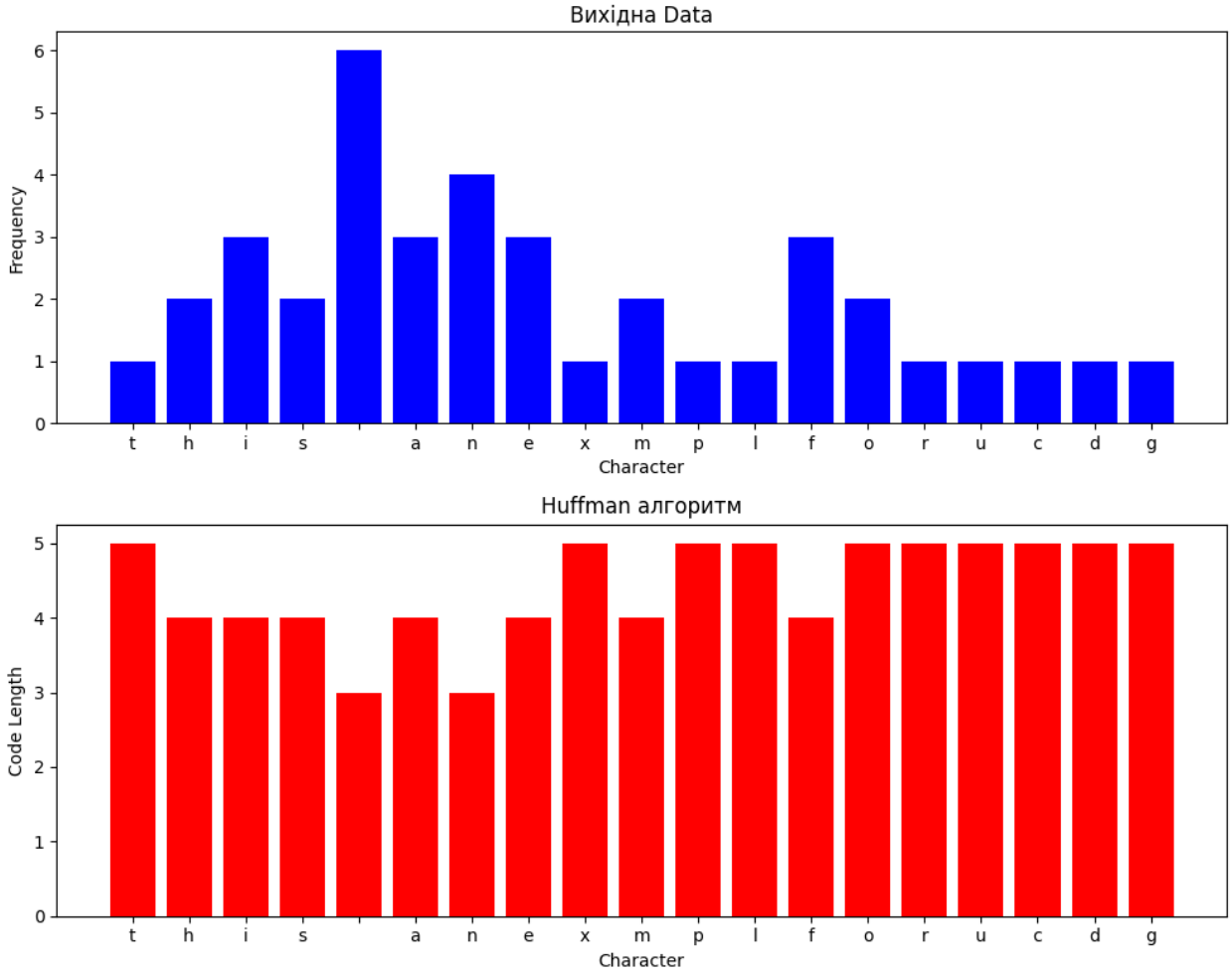


Рисунок 3.4 — Графік результату виконання алгоритму Huffman (виконано самостійно)

Графік показує частоту появи кожного символу у початковому рядку. Вісь X представляє символи. Вісь Y представляє частоту кожного символу, довжину бітового коду для кожного символу.

3.1.3 Алгоритм LZ77 — це метод стиснення даних без втрат, який використовує пошук повторюваних підрядків у вихідних даних. Він працює за допомогою ковзаного вікна, яке зберігає нещодавні дані і використовує їх для пошуку збігів.

Принцип роботи алгоритму полягає у пошуку збігу, для кожного символу у вхідному рядку шукається найдовший підрядок, що збігається з частиною, яка вже була оброблена (в межах ковзаного вікна). Якщо збіг знайдено, він кодується

трійкою (відстань до початку збігу, довжина збігу, наступний символ). Якщо збіг не знайдено, символ кодується трійкою (0, 0, символ).

```python
class LZ77Compressor:
    def __init__(self, window_size=20):
        self.window_size = window_size

    def compress(self, data):
        i = 0
        output = []

        while i < len(data):
            match = self.find_longest_match(data, i)
            if match:
                (best_match_distance, best_match_length) = match
                next_char_index = i + best_match_length
                if next_char_index < len(data):
                    next_char = data[next_char_index]
                else:
                    next_char = ''
                output.append((best_match_distance, best_match_length, next_char))
                i += best_match_length + 1
            else:
                output.append((0, 0, data[i]))
                i += 1

        return output

    def find_longest_match(self, data, current_position):
        end_of_buffer = min(current_position + self.window_size, len(data) + 1)
        best_match_distance = -1
        best_match_length = -1

        for j in range(current_position + 2, end_of_buffer):
            start_index = max(0, current_position - self.window_size)
            substring = data[current_position:j]

            for i in range(start_index, current_position):
                repeat_length = j - current_position
                if data[i:i + repeat_length] == substring:
                    if repeat_length > best_match_length:
                        best_match_distance = current_position - i
                        best_match_length = repeat_length

        if best_match_distance > 0 and best_match_length > 0:
            return (best_match_distance, best_match_length)
        return None

    def decompress(self, compressed_data):
        output = []
        for (distance, length, char) in compressed_data:
            if distance == 0 and length == 0:
                output.append(char)
            else:
                start = len(output) - distance
                for i in range(length):
                    output.append(output[start + i])
                if char:
                    output.append(char)
        return ''.join(output)
```

Рисунок 3.5 — Частина коду алгоритму LZ77 (виконано самостійно)

Вихід за межі рядка додано перевірку для доступу до символу за індексом “i + best_match_length”[3]. Якщо цей індекс перевищує довжину рядка, наступний символ “next_char” буде порожнім рядком . Обробка порожнього символу при

розтисканні виконується за рахунок додавання перевірки, щоб не додавати порожній символ під час розтискання. Розтиснення виконується за рахунок використання трійка для відновлення оригінальних даних, повторюючи збіги та додаючи нові символи. Виконання коду виконано завдяки мові Python.

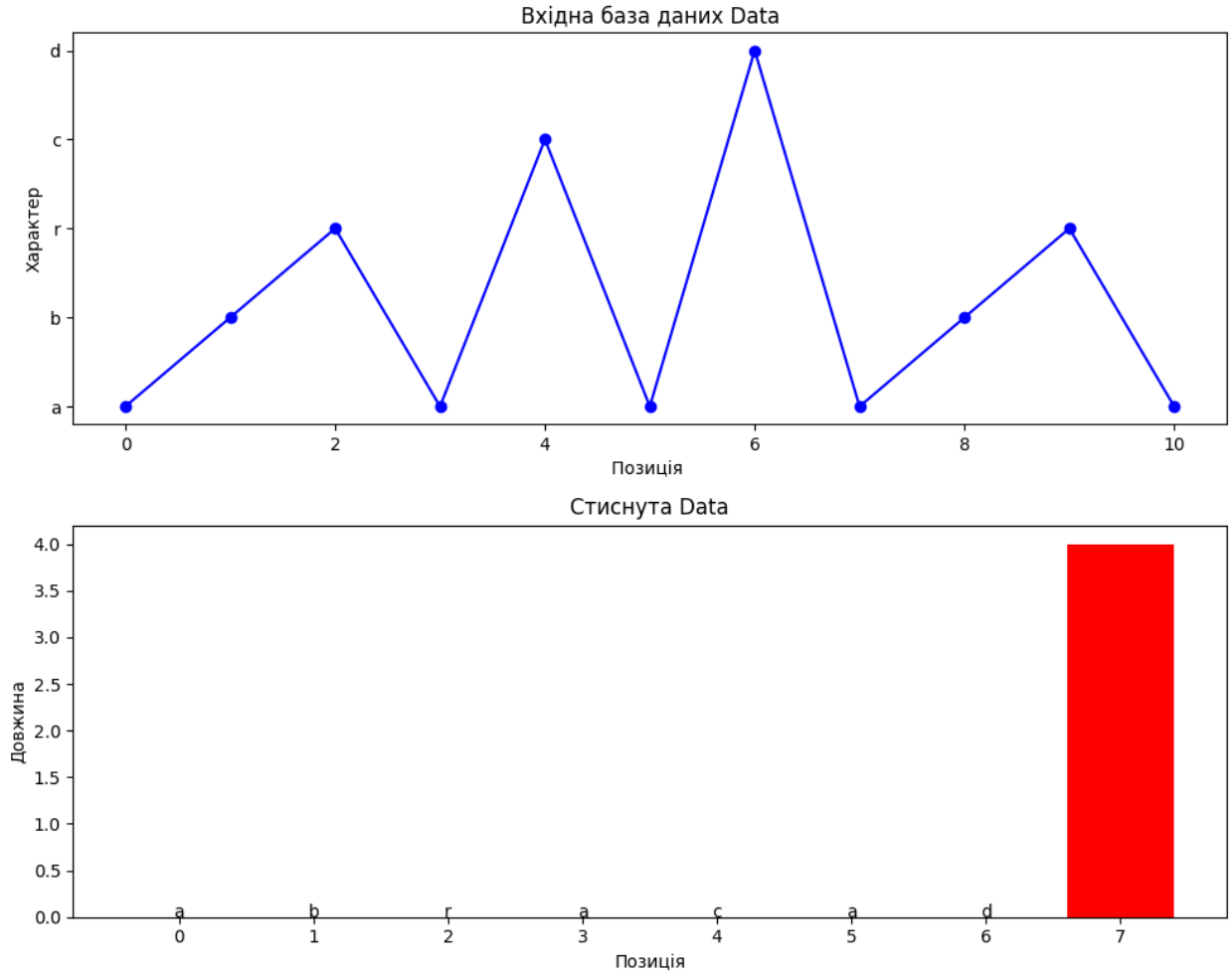


Рисунок 3.6 — Графіки виконання алгоритму LZ77 (виконано самостійно)

На графіку вхідних даних ми бачимо послідовність символів у початковому рядку. Вісь X представляє позиції символів у рядку. Вісь Y представляє самі символи. На графіку стиснених даних можна побачити довжину знайдених збігів на кожній позиції вхідного рядка. Вісь X представляє позиції символів у стисненому рядку. Вісь Y представляє довжину збігу для кожного символу.

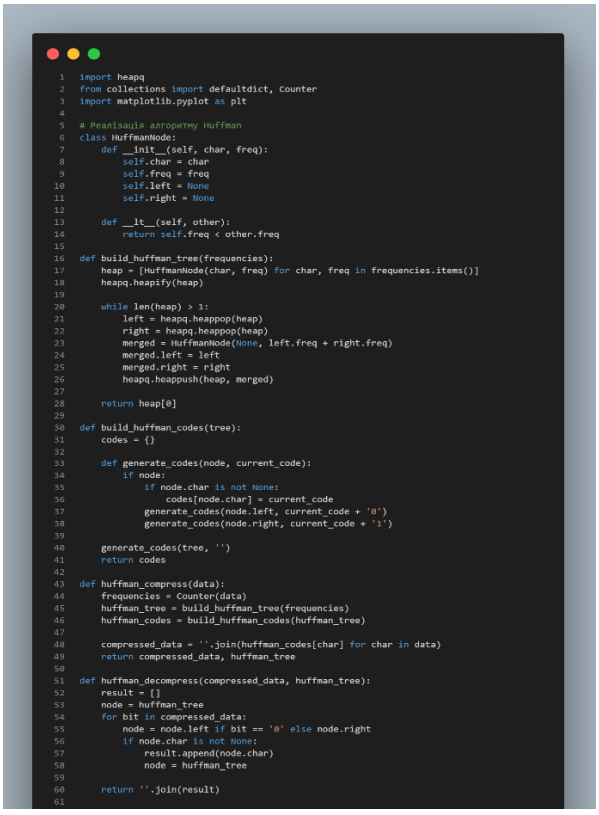
Отже алгоритм LZ77 ефективний для стиснення даних з повторюваними підрядками і широко використовується в різних форматах файлів, таких як ZIP і GZIP.

3.1.4 Подвійний стиск (Binary Compression) — це метод стиснення даних, що використовує дві послідовні фази стиснення для досягнення кращого коефіцієнта стиснення. Найчастіше спочатку застосовують якийсь простий метод стиснення, а потім отриманий результат стискають ще раз іншим методом.

Перша фаза стиснення відбувається коли дані стискаються першим методом (наприклад, алгоритмом Huffman). Друга фаза стиснення відбувається коли отримані дані стискаються другим методом (наприклад, алгоритмом LZ77). Декодування даних проводиться в зворотному порядку, спочатку розпаковуються дані другим методом, а потім першим.

Приклад реалізації алгоритму Подвійного стиску на Python алгоритм. Як приклад було обрано речення: “Nure is the best university in Kharkiv”.

Першим етапом виконуємо алгоритм стиснення Huffman. Виконується частотний аналіз даних та побудова дерева “Huffman” з кодуванням символів за допомогою отриманого дерева.



```python
import heapq
from collections import defaultdict, Counter
import matplotlib.pyplot as plt

# Реалізація алгоритму Huffman
class HuffmanNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(frequencies):
    heap = [HuffmanNode(char, freq) for char, freq in frequencies.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]

def build_huffman_codes(tree):
    codes = {}

    def generate_codes(node, current_code):
        if node:
            if node.char is not None:
                codes[node.char] = current_code
            generate_codes(node.left, current_code + '0')
            generate_codes(node.right, current_code + '1')

    generate_codes(tree, '')
    return codes

def huffman_compress(data):
    frequencies = Counter(data)
    huffman_tree = build_huffman_tree(frequencies)
    huffman_codes = build_huffman_codes(huffman_tree)

    compressed_data = ''.join(huffman_codes[char] for char in data)
    return compressed_data, huffman_tree

def huffman_decompress(compressed_data, huffman_tree):
    result = []
    node = huffman_tree
    for bit in compressed_data:
        node = node.left if bit == '0' else node.right
        if node.char is not None:
            result.append(node.char)
            node = huffman_tree

    return ''.join(result)
```

Рисунок 3.7 — Реалізація алгоритму коду алгоритму Huffman (виконано самостійно)

Другий етап реалізація алгоритма LZ77[10]. Алгоритм LZ77виконує пошук найдовших збігів у даних котрі були отримані в результаті стиснення алгоритму Huffman. Кодування збігів та залишкових символів. Рисунок 3.8 зображена друга частина, реалізація коду LZ77.

```python
# Реалізація алгоритму LZ77
class LZ77Compressor:
    def __init__(self, window_size=20):
        self.window_size = window_size

    def compress(self, data):
        i = 0
        output = []

        while i < len(data):
            match = self.find_longest_match(data, i)
            if match:
                (best_match_distance, best_match_length) = match
                next_char_index = i + best_match_length
                if next_char_index < len(data):
                    next_char = data[next_char_index]
                else:
                    next_char = ''
                output.append((best_match_distance, best_match_length, next_char))
                i += best_match_length + 1
            else:
                output.append((0, 0, data[i]))
                i += 1

        return output

    def find_longest_match(self, data, current_position):
        end_of_buffer = min(current_position + self.window_size, len(data) + 1)
        best_match_distance = -1
        best_match_length = -1

        for j in range(current_position + 2, end_of_buffer):
            start_index = max(0, current_position - self.window_size)
            substring = data[current_position:j]

            for i in range(start_index, current_position):
                repeat_length = j - current_position
                if data[i:i + repeat_length] == substring:
                    if repeat_length > best_match_length:
                        best_match_distance = current_position - i
                        best_match_length = repeat_length

        if best_match_distance > 0 and best_match_length > 0:
            return (best_match_distance, best_match_length)
        return None

    def decompress(self, compressed_data):
        output = []
        for (distance, length, char) in compressed_data:
            if distance == 0 and length == 0:
                output.append(char)
            else:
                start = len(output) - distance
                for i in range(length):
                    output.append(output[start + i])
                if char:
                    output.append(char)
        return ''.join(output)

# Функція для візуалізації
def visualize_double_compression(data, huffman_compressed_data, lz77_compressed_data):
    fig, axs = plt.subplots(3, 1, figsize=(10, 12))

    # Оригінальні дані
    axs[0].bar(range(len(data)), list(map(ord, data)), color='b')
    axs[0].set_title('Вихідна Data')
    axs[0].set_xlabel('Position')
    axs[0].set_ylabel('Character Code')

    # Стиснені дані (Huffman)
    axs[1].bar(range(len(huffman_compressed_data)), list(map(int, huffman_compressed_data)), color='g')
    axs[1].set_title('стеснена методом Huffman Data')
    axs[1].set_xlabel('Position')
    axs[1].set_ylabel('Bit')

    # Стиснені дані (Huffman + LZ77)
    positions = []
    lengths = []
    chars = []

    for (distance, length, char) in lz77_compressed_data:
        positions.append(distance)
        lengths.append(length)
        chars.append(char)

    axs[2].bar(range(len(lz77_compressed_data)), lengths, color='r')
    axs[2].set_title('Huffman + LZ77 стеснення Data')
    axs[2].set_xlabel('Position')
    axs[2].set_ylabel('Length')

    for i, char in enumerate(chars):
        axs[2].text(i, lengths[i], char, ha='center')

    plt.tight_layout()
    plt.show()

# Приклад використання Подвійного стиску
data = "Nure is th best university in Kharkiv"

# Перша фаза стиснення - Huffman
huffman_compressed_data, huffman_tree = huffman_compress(data)

# Друга фаза стиснення - LZ77
lz77_compressor = LZ77Compressor(window_size=20)
lz77_compressed_data = lz77_compressor.compress(huffman_compressed_data)

# Візуалізація
visualize_double_compression(data, huffman_compressed_data, lz77_compressed_data)
```

Рисунок 3.8 — Реалізація коду алгоритму LZ77 (виконано самостійно)

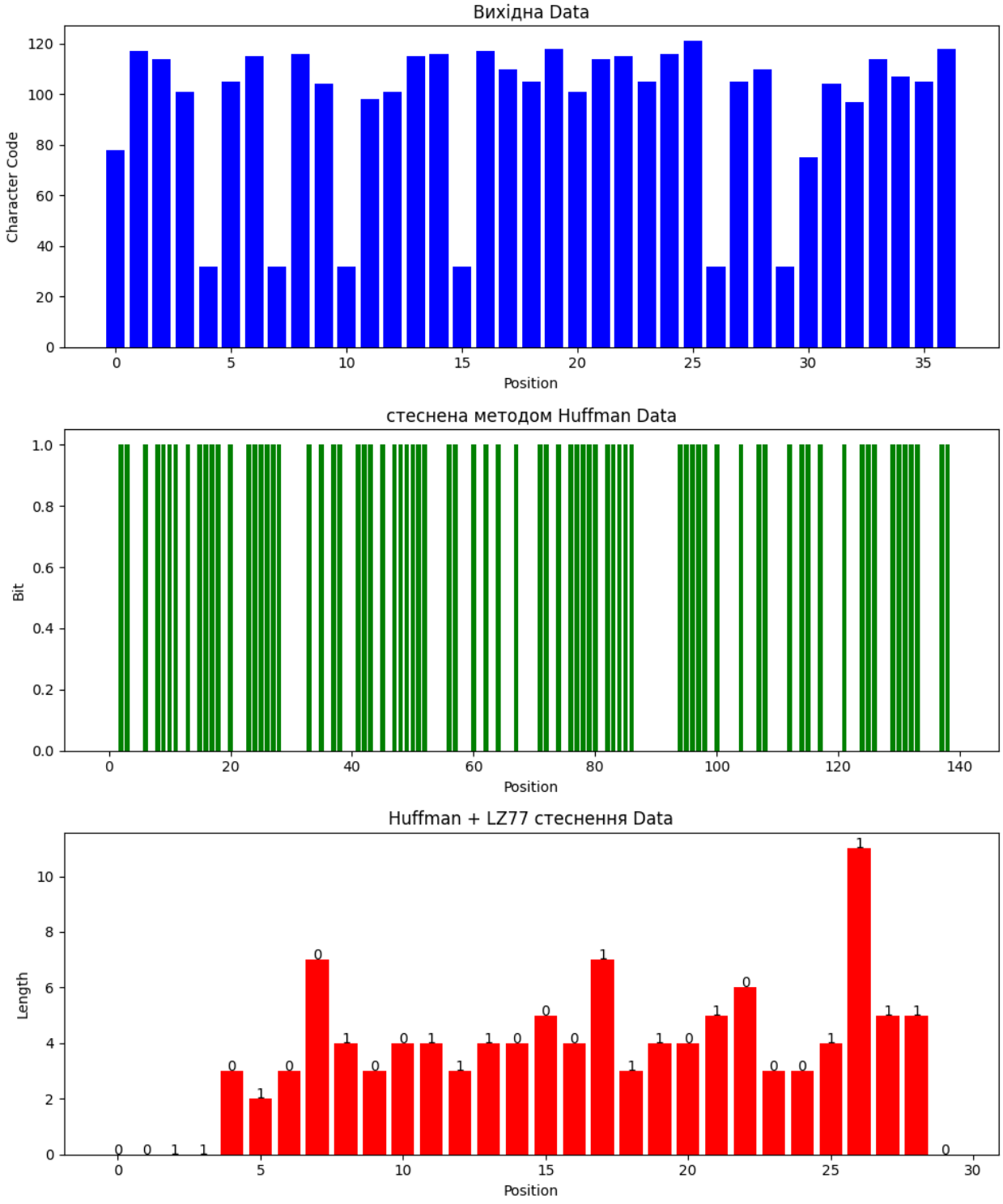


Рисунок 3.9 — Графік отриманих результатів Binary compression (виконано самостійно)

Перший графік (Вихідна Data), показує вихідні дані у вигляді стовпчикової діаграми, де по осі x розташовані позиції символів, а по осі y — коди символів (ASCII або Unicode). Другий графік (стиснення Huffman Data), показує стиснені дані за допомогою алгоритму Huffman у вигляді стовпчикової діаграми, де кожен

стовпчик представляє біт стиснених даних. Третій графік (Huffman + LZ77 стиснення Data),п оказує результати подальшого стиснення даних алгоритмом LZ77 після стиснення Huffman. На осі x розташовані позиції стиснених блоків, а на осі y — довжини знайдених збігів у вікні. Кожен блок також містить символ, який додається після збігу.

Отже метод подвійного стиску поєднує переваги двох різних методів стиснення для досягнення кращого коефіцієнта стиснення. Спочатку дані стискаються за допомогою алгоритму Huffman, потім стискаються ще раз алгоритмом LZ77. Декодування проводиться в зворотному порядку. Цей підхід дозволяє ефективніше зберігати дані з високою редукцією розміру.

3.1.5 Алгоритм Deflate[17] є комбінацією двох інших методів стиснення алгоритму LZ77 та алгоритму Huffman. Спочатку він використовує LZ77 для заміни повторюваних рядків даних посиланнями, а потім використовує коди Huffman для подальшого стискання вихідних даних. У результаті Deflate забезпечує ефективне стиснення та широко використовується у багатьох форматах файлів, таких як ZIP та gzip.

Принцип роботи алгоритма Deflate[17] наступний, LZ77-компресія відбувається по принципу дані розділяються на послідовності символів, і кожна повторювана послідовність замінюється трійкою (відстань, довжина, наступний символ). Кодування Huffman відбувається при отримані послідовності, кодуються за допомогою алгоритму Huffman, що забезпечує додаткове стиснення.

Приклад реалізації алгоритму Deflate на Python з візуалізацією результатів стиснення.

```python
import zlib
import matplotlib.pyplot as plt

# Функція для стиснення за допомогою алгоритму Deflate
def deflate_compress(data):
    return zlib.compress(data.encode())

# Функція для розтиснення за допомогою алгоритму Deflate
def deflate_decompress(compressed_data):
    return zlib.decompress(compressed_data).decode()

# Функція для візуалізації
def visualize_deflate_compression(data, compressed_data):
    fig, axs = plt.subplots(2, 1, figsize=(10, 8))

    # Оригінальні дані
    axs[0].bar(range(len(data)), list(map(ord, data)), color='b')
    axs[0].set_title('Вихідні дані')
    axs[0].set_xlabel('Position')
    axs[0].set_ylabel('Character Code')

    # Стиснені дані
    axs[1].bar(range(len(compressed_data)), list(compressed_data), color='g')
    axs[1].set_title('Deflate стиснення Data')
    axs[1].set_xlabel('Position')
    axs[1].set_ylabel('Byte Value')

    plt.tight_layout()
    plt.show()

# Приклад використання алгоритму Deflate
data = "Nure is the best university in Kharkiv" * 10  # Повторювані дані для кращої демонстрації стиснення

# Стиснення даних
compressed_data = deflate_compress(data)
print("Compressed data:", compressed_data)

# Розтиснення даних
decompressed_data = deflate_decompress(compressed_data)
print("Decompressed data:", decompressed_data)

# Перевірка на коректність розтиснення
assert decompressed_data == data

# Візуалізація
visualize_deflate_compression(data, compressed_data)

```

Рисунок 3.10 — Реалізація алгоритму коду алгоритму Deflate (виконано самостійно)

Спершу дані проходять через алгоритм LZ77, який знаходить і замінює повторювані послідовності символів. Кожна повторювана послідовність замінюється трійкою (відстань, довжина, наступний символ). Відстань — це відстань від поточного символу до попереднього збігу. Довжина — це довжина збігу. Наступний символ — це символ, який йде після збігу. Наступним відбувається кодування Huffman, отримані трійки кодуються за допомогою

алгоритму Huffman. Алгоритм створює дерево Huffman на основі частот символів, які зустрічаються в даних. Символи з більш високою частотою отримують коротші коди, що забезпечує ефективне стиснення. Декодування відбувається шляхом розтиснення даних здійснюється у зворотному порядку, спочатку розкодовуються коди Huffman, а потім за допомогою LZ77 відновлюються оригінальні дані. Як приклад був використан той же самий вислів ” Nure is the best university in Khrkiv”. Код виконано на мові Python котрий зображений на рисунок 3.10.

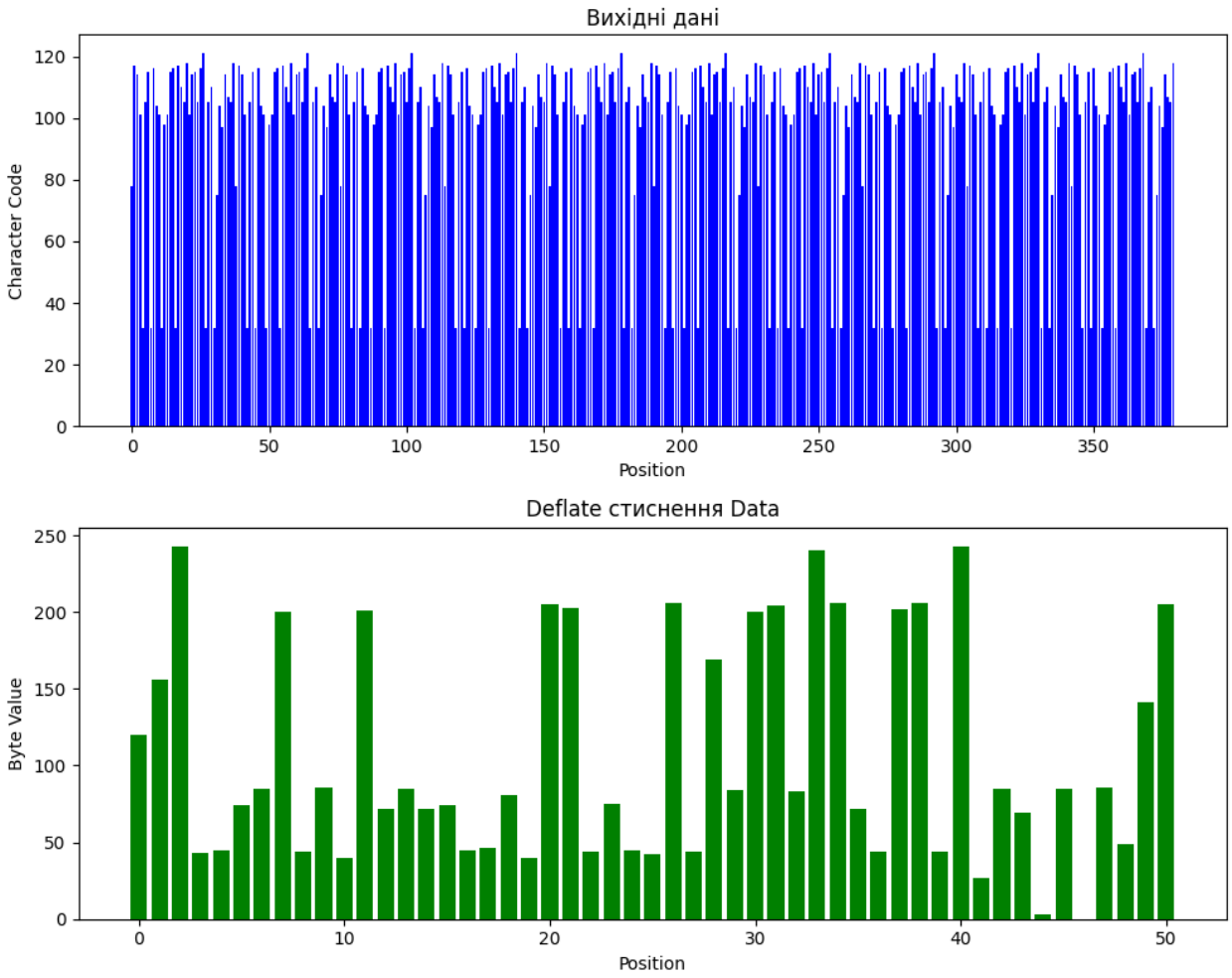


Рисунок 3.11 — Графік результату алгоритму стиснення Deflate (виконано самостійно)

Отже на першому графіку “Вихідні дані”, показано вихідні дані у вигляді стовпчикової діаграми, де по осі x розташовані позиції символів, а по осі y — коди символів (ASCII або Unicode). На другий графік “Deflate стиснення Data” показує

стиснені дані за допомогою алгоритму Deflate у вигляді стовпчикової діаграми, де по осі x розташовані позиції байтів у стиснених даних, а по осі y — значення байтів.

Висновок можна зробити наступний, що алгоритм Deflate є популярним методом стиснення даних, який поєднує дві техніки, кодування LZ77 та кодування Huffman. Він використовується в таких форматах стиснення, як ZIP, gzip та PNG.

3.1.6 Алгоритм Run-Length Encoding (RLE)[1] стискує дані, зменшуючи послідовності повторюваних символів. Він замінює кожну послідовність повторюваних символів парою (символ, кількість повторень). Цей алгоритм особливо ефективний для стиснення даних, які містять довгі послідовності одного й того ж символу.

Принцип роботи алгоритму RLE складається з трьох кроків. Перший крок це прочитання послідовностей вхідних символів. Другим етапом визначення кількості повторень поточних символів. Третій етап це запис символів та кількість їх повторень.

Розглянемо приклад реалізації алгоритму RLE на Python [3]зображено на рисунку 3.12, а також візуалізація результатів стиснення. Як вихідні дані приймаємо все той же вислів:”Nure is the best university in Kharkiv”. Після виконання коду отримали наступний графік результату зображено на рисунку 3.12.

```python
import matplotlib.pyplot as plt

# Функція для стиснення за допомогою алгоритму RLE
def rle_compress(data):
    compressed = []
    i = 0
    while i < len(data):
        count = 1
        while i + 1 < len(data) and data[i] == data[i + 1]:
            i += 1
            count += 1
        compressed.append((data[i], count))
        i += 1
    return compressed

# Функція для розтиснення за допомогою алгоритму RLE
def rle_decompress(compressed_data):
    decompressed = []
    for char, count in compressed_data:
        decompressed.append(char * count)
    return ''.join(decompressed)

# Функція для візуалізації
def visualize_rle_compression(data, compressed_data):
    fig, axs = plt.subplots(2, 1, figsize=(10, 8))

    # Оригінальні дані
    axs[0].bar(range(len(data)), list(map(ord, data)), color='b')
    axs[0].set_title('Вихідні дані')
    axs[0].set_xlabel('Position')
    axs[0].set_ylabel('Character Code')

    # Стиснені дані
    compressed_values = []
    for char, count in compressed_data:
        compressed_values.extend([ord(char)] * count)
    axs[1].bar(range(len(compressed_values)), compressed_values, color='g')
    axs[1].set_title('стистнуті дані алгоритмом RLE')
    axs[1].set_xlabel('Position')
    axs[1].set_ylabel('Character Code')

    plt.tight_layout()
    plt.show()

# Приклад використання алгоритму RLE
data = "Nure is the best university in Kharkiv"  # Приклад даних

# Стиснення даних
compressed_data = rle_compress(data)
print("Стиснута data:", compressed_data)

# Розтиснення даних
decompressed_data = rle_decompress(compressed_data)
print("Рестиснута data:", decompressed_data)

# Перевірка на коректність розтиснення
assert decompressed_data == data

# Візуалізація
visualize_rle_compression(data, compressed_data)
```

Рисунок 3.12 — Реалізація алгоритму коду алгоритму RLE (виконано самостійно)

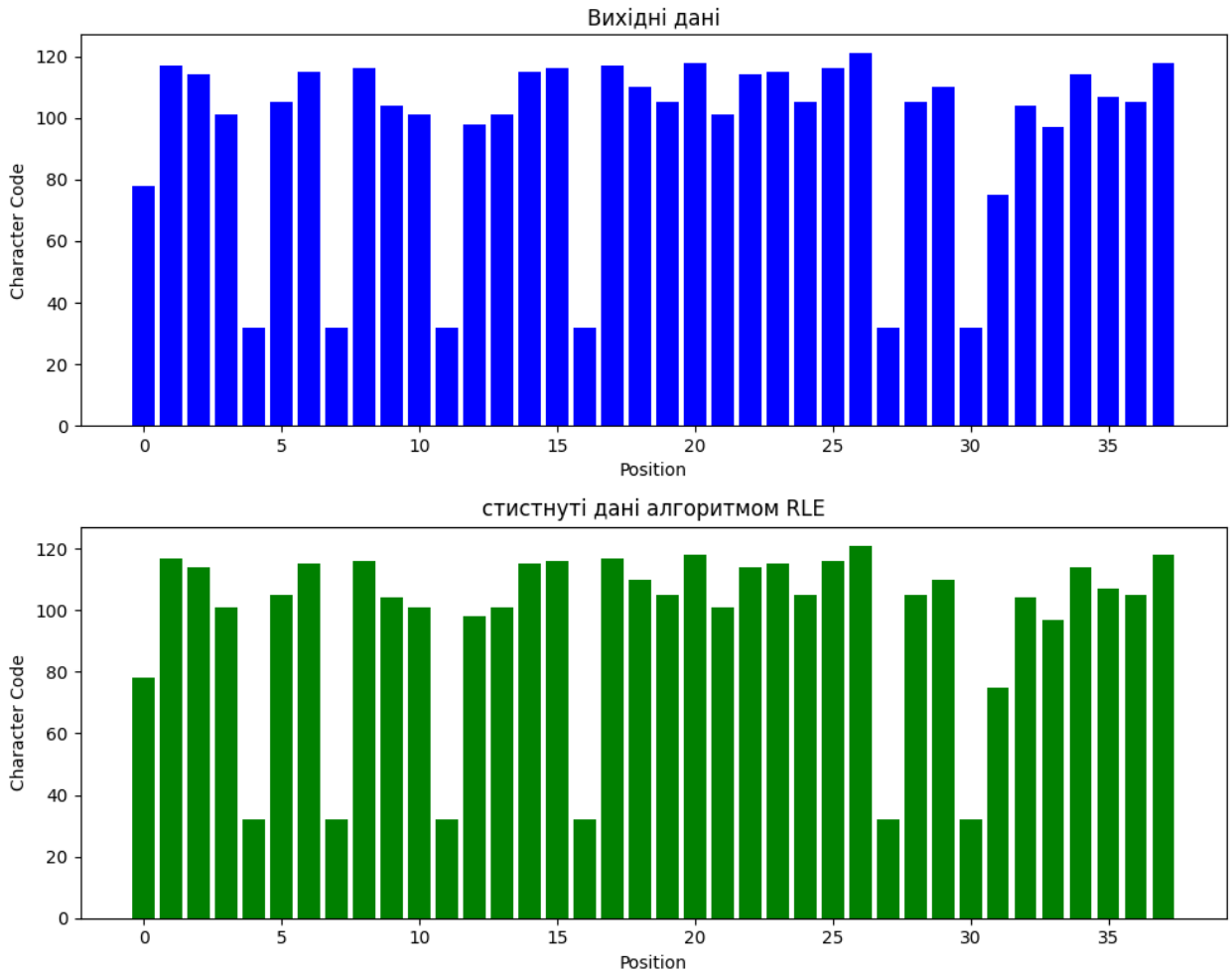


Рисунок 3.13 — Графік результатів виконання алгоритму RLE (виконано самостійно)

Отже перший графік показує вихідні дані у вигляді стовпчикової діаграми, де по осі x розташовані позиції символів, а по осі y — коди символів (ASCII або Unicode). Другий графік нам показу, що стиснені дані за допомогою алгоритму RLE у вигляді стовпчикової діаграми, де по осі x розташовані позиції байтів у стиснених даних, а по осі y — значення байтів.

Розтиснення виконується для кожної пари (символ, кількість повторень) у стиснених даних, спершу відновлюється послідовність символів на основі кількості повторень, а наступним об'єднуються всі відновлені символи в оригінальну послідовність.

Підводячи підсумок, RLE є корисним інструментом стиснення для специфічних випадків використання, таких як графічні зображення з великими

однотонними областями або текстові дані з багатьма повторюваними символами. Однак для загального стиснення даних, особливо для різноманітних і випадкових даних, слід розглянути використання більш універсальних алгоритмів стиснення, таких як LZ77, Deflate або алгоритм Huffman.

4. РЕЛЯЦІЙНА БАЗА ДАНИХ

4.1 Обирання виду бази даних

Мій проект передбачує база даних для місцевої лікарні в котрій присутні різ фахівці та відділення. До бази даних проєкта буде входити база даних лікарів, пацієнтів(амбулаторна карта), рецепти на препарати, препарати та їх опис, та прийом пацієнта. Для проєкта потрібно обрати архітектуру та модельний метод. Як модельний метод було обрано реляційну базу даних. А архітектуру для неї багаторівневу.

Реляційна база даних є відмінним вибором для лікарні з наступних причин:

Структурованість даних, у лікарні інформація про пацієнтів, медичні записи, результати тестів та інші дані мають структурований формат, який можна легко представити у вигляді таблиць з реляційною базою даних.

Складні запити та зв'язки, реляційні бази даних дозволяють легко виконувати складні запити, такі як звіти про здоров'я пацієнтів, аналізи ефективності лікування та інші аналітичні завдання, які можуть бути важливими для лікарні.

Забезпечення цілісності даних адже реляційні бази даних надають механізми для забезпечення цілісності даних, такі як обмеження цілісності, транзакції та відновлення в разі відмови, що дуже важливо для медичних даних.

Безпека даних у реляційних базах даних присутні механізми автентифікації, авторизації та контролю доступу, що дозволяє забезпечити конфіденційність та безпеку медичних даних.

Масштабовність адже сучасні реляційні бази даних мають можливості для горизонтального масштабування та реплікації, що дозволяє забезпечити високу доступність та продуктивність системи, навіть при великому обсязі даних.

Усі ці функції та можливості реляційних баз даних роблять їх відмінним вибором для лікарень, де потрібно забезпечити ефективне зберігання, управління та аналіз медичних даних.

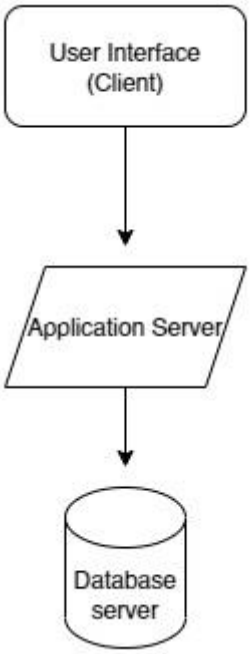


Рисунок 4.1 — Блок схема багаторівнева архітектура (виконано самостійно)

Була розроблена блок схема для бази даних. База даних повинна мати наступні таблиці: персоналу, прийому пацієнтів, назву препаратів та їх опис, рецепт на препарат від лікаря. блок схема була зроблена наступна

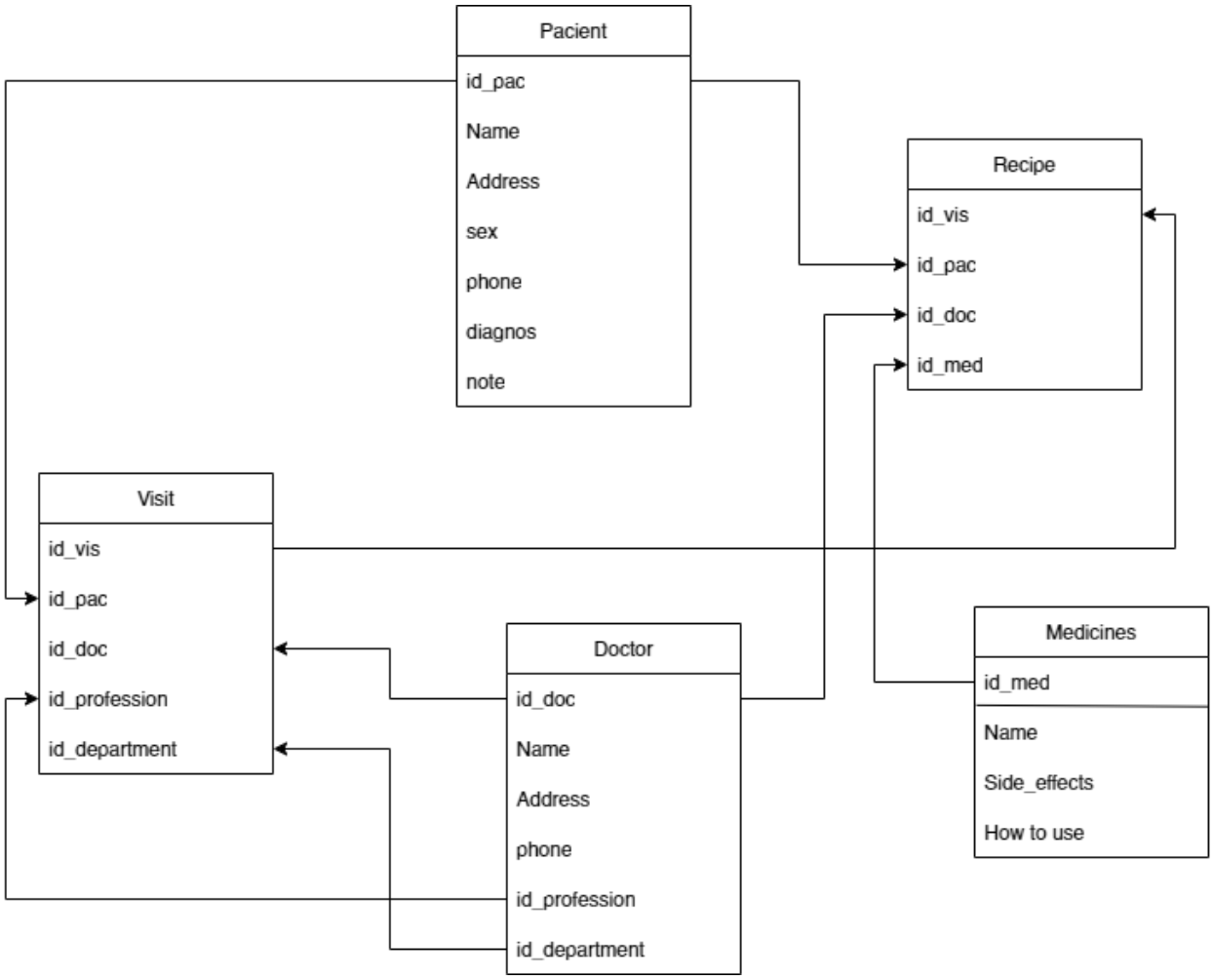


Рисунок 4.2 — Блок схема реляційної бази даних (виконано самостійно)

Були сформовані наступні таблиці де id_doc, id_pac, id_med, id_vis, де мають унікальні “ID” значення. Таблиці були створені наступним чином, розглянемо приклад з таблицею Лікарі “Doctor”, фрагмент програмного коду з використанням мови SQL інтерпретованої у мову Python, фреймворк Django.

```python
class Doctors(models.Model):
    FirstName = models.CharField(max_length=20)
    LastName = models.CharField(max_length=35)
    Address = models.TextField(blank=True)
    Phone = models.CharField(max_length=13)

    PR = "PR"
    IN = "IN"
    Medsis = "Medsis"
    DC = "DC"

    doctorsChoices = (
        (PR, "Professor"),
        (IN, "Intern"),
        (Medsis, "Nurse"),
        (DC, "Doctor"),
    )

    Profession = models.CharField(
        choices=doctorsChoices,
        default=DC,
        blank=True,
        max_length=10
    )

    NEUROLOGY = "Neurology"
    CARDIOLOGY = "Cardiology"
    ORTHOPEDICS = "Orthopedics"
    GYNECOLOGY = "Gynecology"

    departmentChoices = (
        (NEUROLOGY, "Neurology"),
        (CARDIOLOGY, "Cardiology"),
        (ORTHOPEDICS, "Orthopedics"),
        (GYNECOLOGY, "Gynecology"),
    )
```

Рисунок 4.1 — Створення таблиці “Doctor” (виконано самостійно)

З коду можна побачити “doc_id” ми використовуємо “Primary Key autoincrement” він встановлюється default [23]тобто кожний “id” є унікальний та кожний наступний “id” збільшується на одиницю. Показник “FirstName” та “LastName”було зроблено текстом та довжина щоб не перевищувала для імя 20 символів, а для прізвища не більше 35 тобто поле неможливо залишити порожнім, щоб у подальшому не виникало помилок у пошуку персоналу. Таку ж само операцію було зроблено з стовбцями “Address”, та у стовбцях “Phone”, profession, department, пустих показників не може бути отже за (default) за замовчуванням встановлюємо DC тобто лікар, такаж само операція і з департаментом.

Після виконання та дотримання всіх обмежень отримав наступну базу даних.

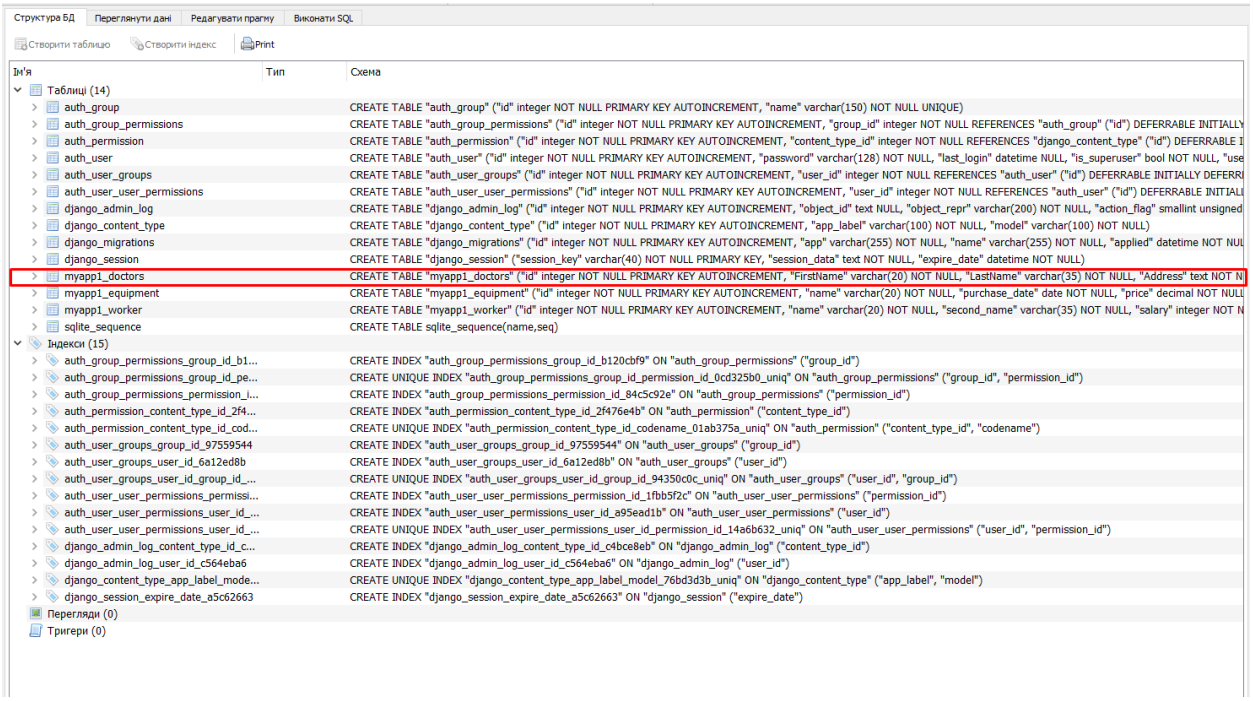


Рисунок 4.2 — Зміст бази даних та атрибути таблиць (виконано самостійно)

Так як Web проект було розроблено у фреймворку Django, також по створювалися додаткові таблиці для проекту. Мова написання у фреймворку Django, python. Фреймворк Django було обрано бо завдяки ньому створюється Web сервер, завдяки якому можна співпрацювати з Front-End частиною, та Back-End тобто БД та багато іншого, нижче наведено взаємодію проекта у цілому[23]. Завдяки Django реалізується архітектура багаторівнева.

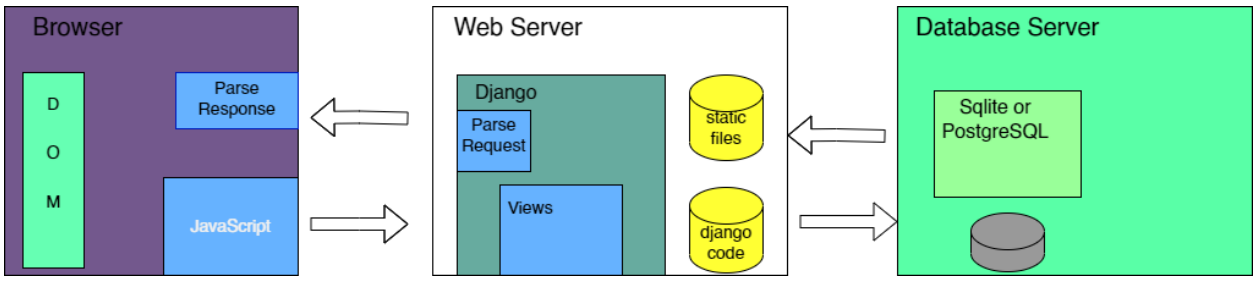


Рисунок 4.3 — Принцип роботи Django проекту[20]

Додавати лікарів у базу можна і завдяки панелі адміністратора, а на пряму через код. Вхід в панель адміністратора виконується завдяки логіну та паролю, так як додавання до бази є вузьким колом людей, то облікові записи додаються власноруч, головним адміністратором.

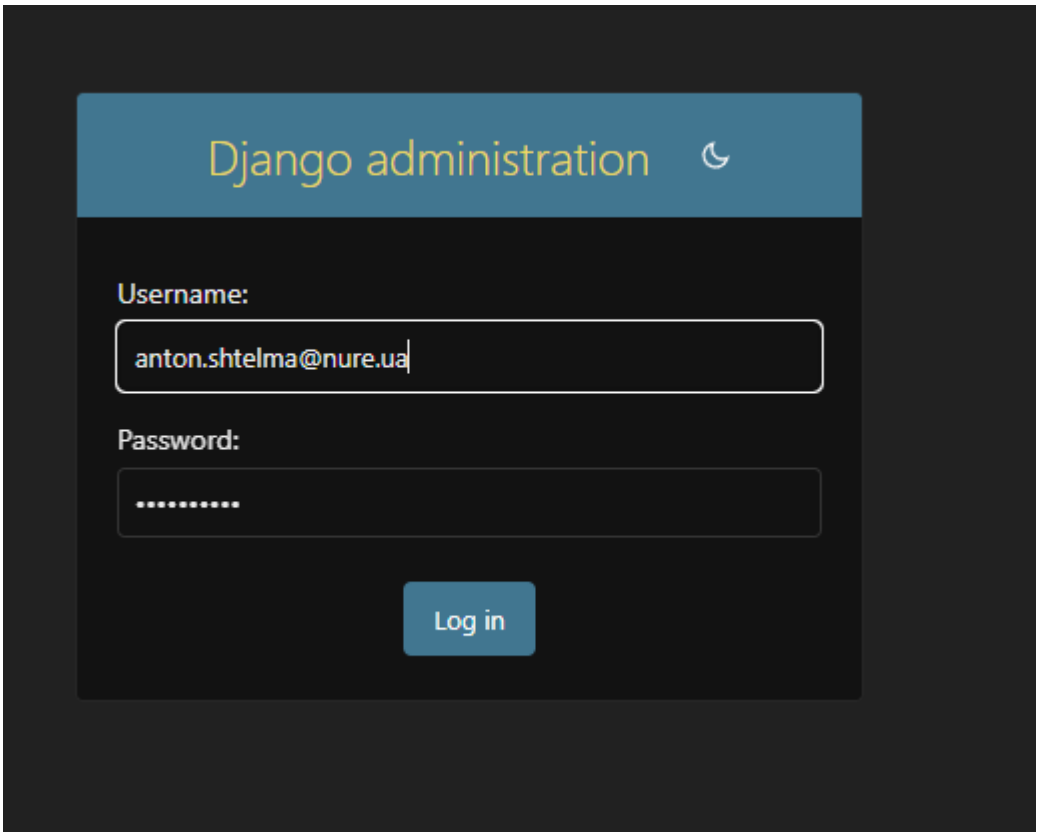


Рисунок 4.4 — Вхід до обліковий запис (виконано самостійно)

Після входу в обліковий запис ми потрапляємо до Dashbord. Він виглядає наступним чином

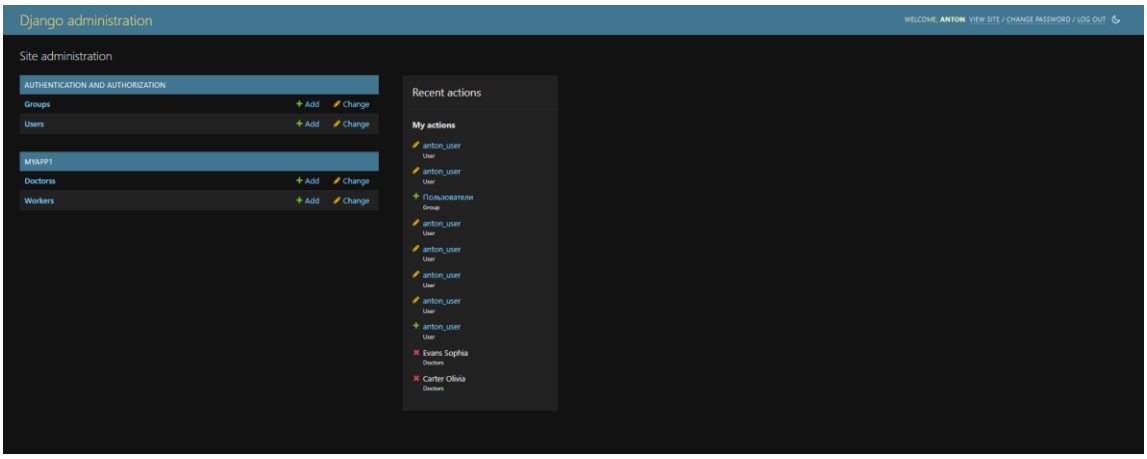


Рисунок 4.5 — Dashbord облікового запису (виконано самостійно)

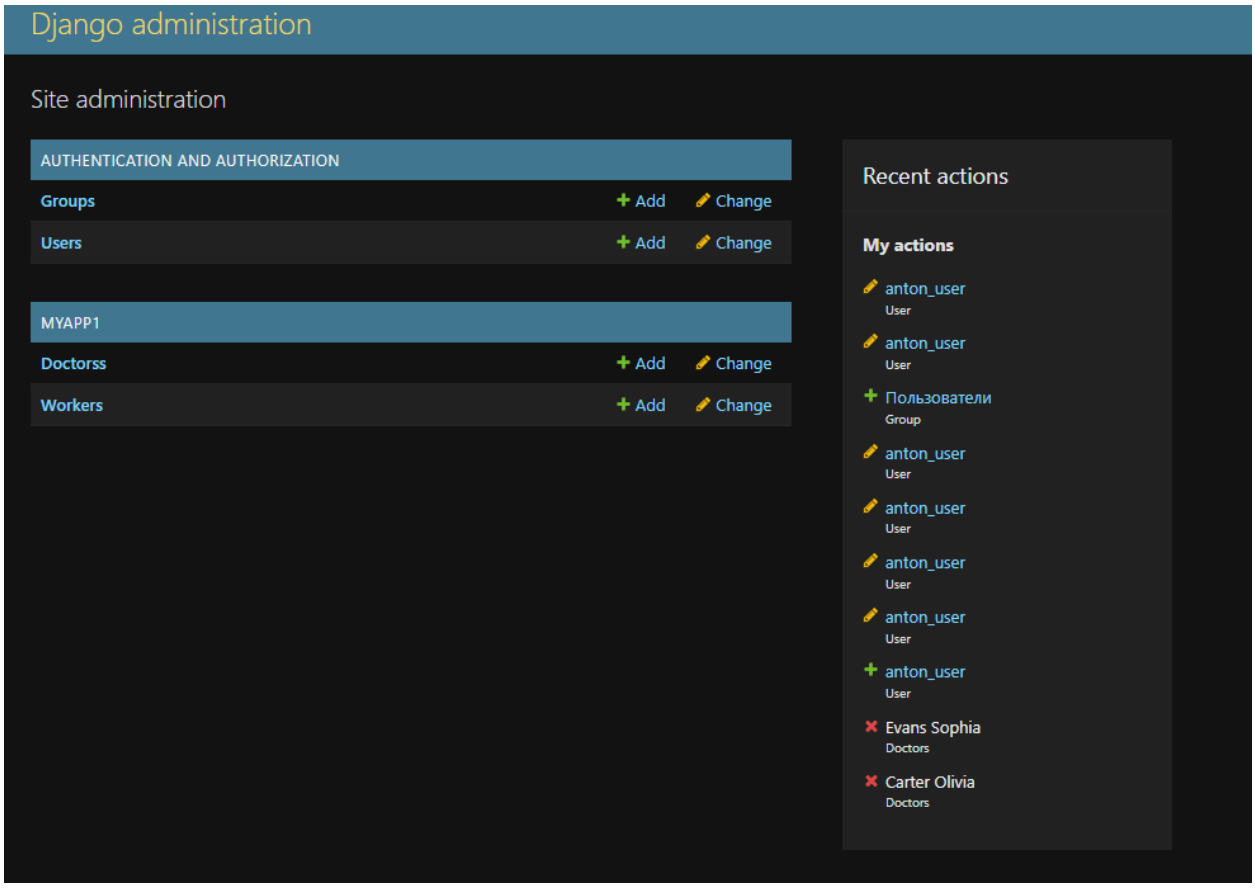


Рисунок 4.6 — Dashbord облікового запису ближче (виконано самостійно)

На цьому скрині можна побачити таблицю Doctors, “Users” це відвідувачі лікарні, та їх облікові дані. Так як це головний адміністратор, то є можливість поділити усіх користувачів за групами, та надати перелік необхідних функцій, наприклад медсестра з регістратури, буде надано доступ до перегляду таблиці пацієнтів, докторів, а наприклад у таблиці “Visit” можливість записувати на прийом пацієнтів до певного лікаря. У відділа кадрів буде наданий доступ до редагування працівників, тобто докторів, матимуть вони наступний інтерфейс.

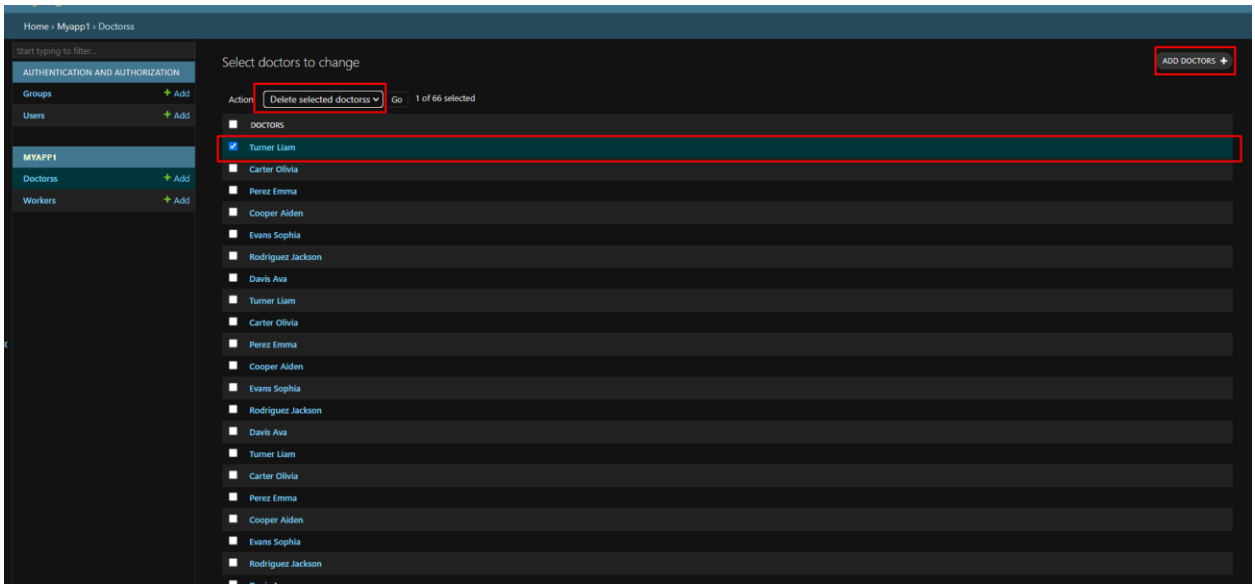


Рисунок 4.7 — Інтерфейс таблиці Doctors (виконано самостійно)

У цьому вікні можна додавати, або видаляти докторів, а якщо натиснути на певне прізвище та ім’я, то можна редагувати інформацію.

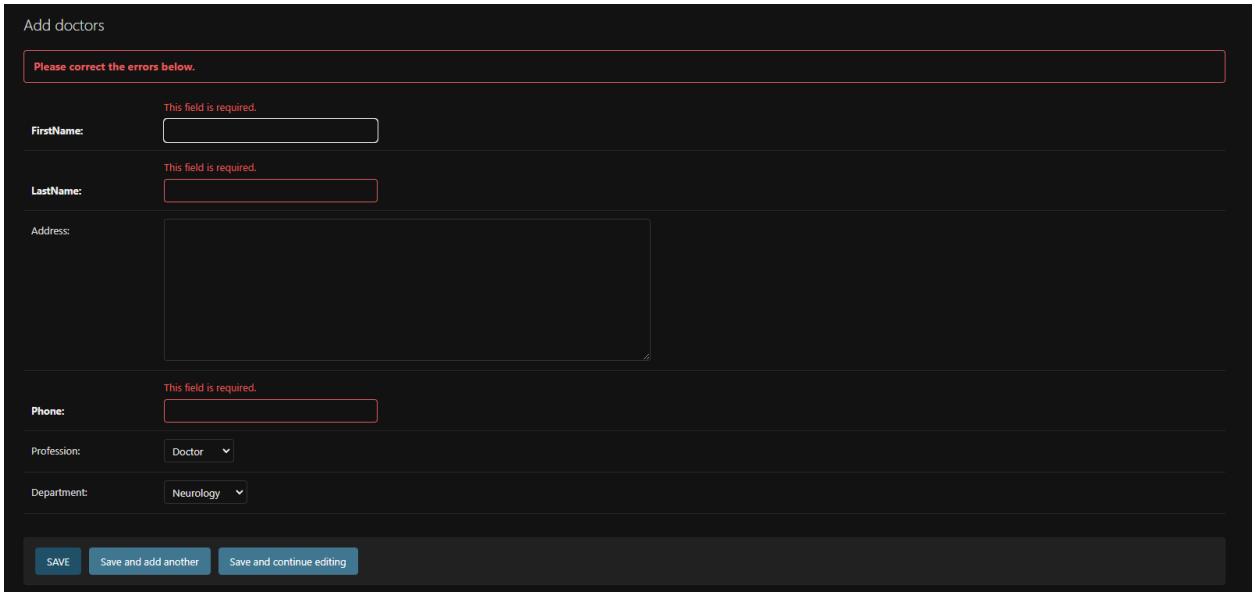


Рисунок 4.8 — Додавання нового лікаря до таблиці Doctors (виконано самостійно)

Як можна побачити з рисунку, неможливо додати лікаря, попередньо не заповнивши вільні місця. Додавання професії та департамент треба обрати з варіантів.

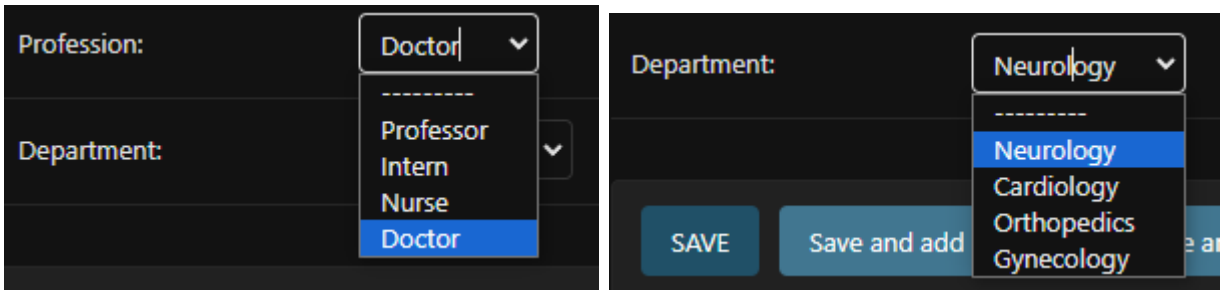


Рисунок 4.9 — Обирання прфоесії та департаменту (виконано самостійно)

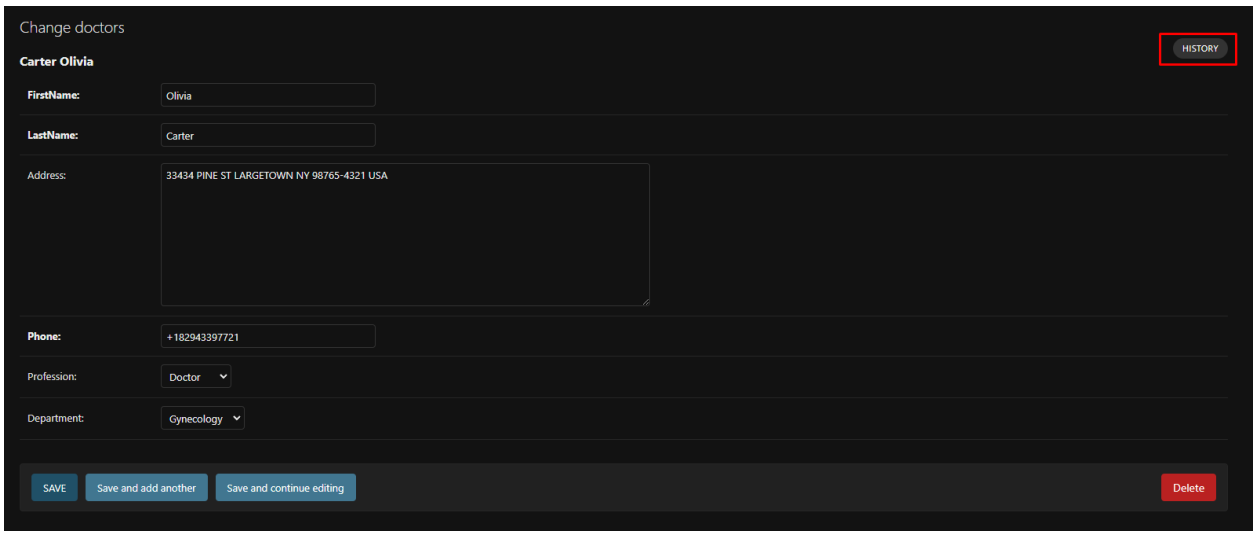


Рисунок 4.10 — Редагування даних про лікаря (виконано самостійно)

Також присутня можливість редагування лікарів, у разі зміни адреси лікаря чи номеру телефону. Та розділ “History” у подальшому дасть змогу переглядати кількість пацієнтів котрі прийшли до лікарні на прийом.

Додавання лікарів також можна виконувати сетом завдяки коду, проте потрібно чітко вказати усі потрібні атрибути запису, зображено нижче.

Рисунок 4.11 — Додавання лікарів використовуючи мову програмування Python (виконано самостійно)

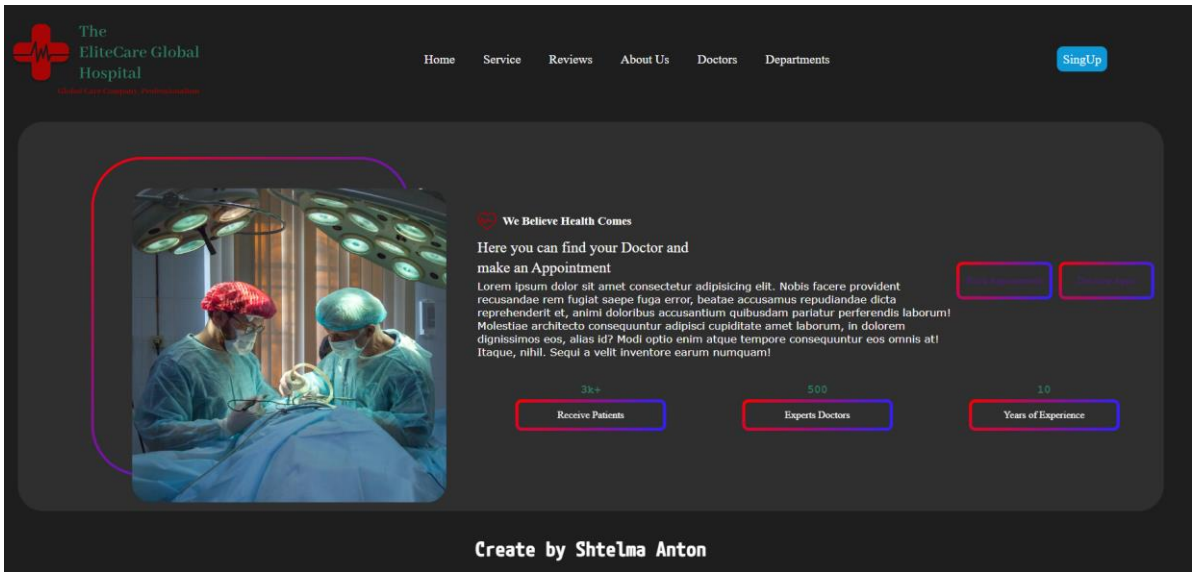


Рисунок 4.12 — Головна сторінка клініки (виконано самостійно)

На головному сайті клініки можна, авторизуватися за власним обліковим записом, або якщо його немає саме у цій клініці то можна переглянути лікарів, а саме їх прізвище та ім’я та де вони працюють у якому департаменті.

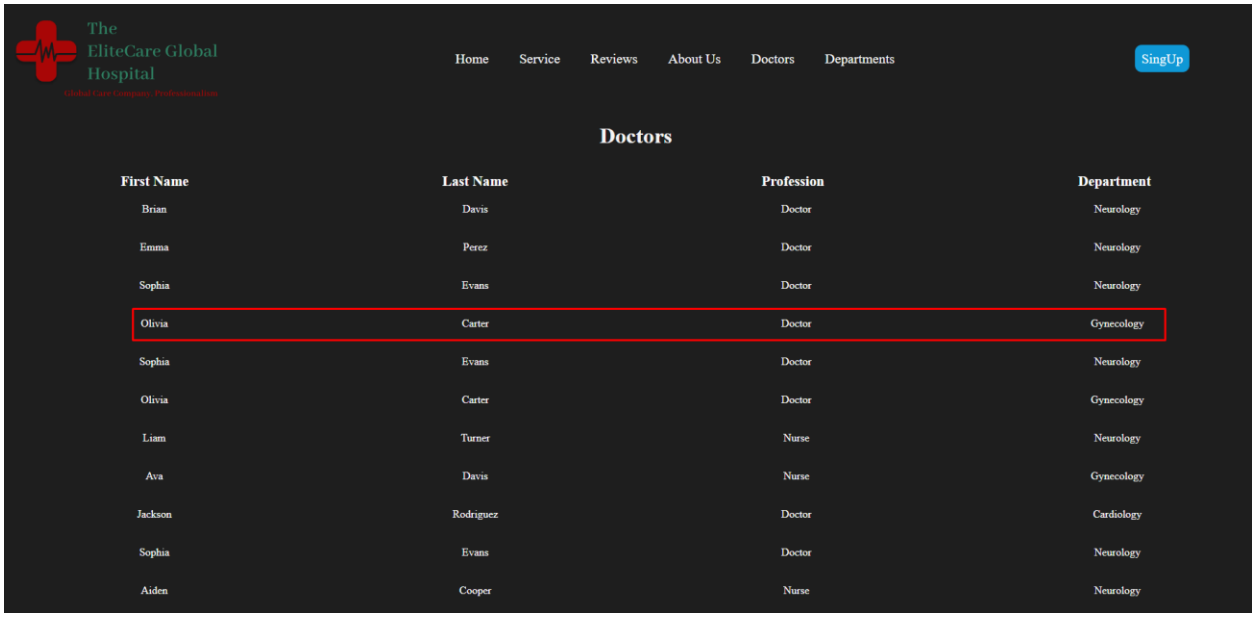


Рисунок 4.13 — розділ сторінки “Doctors”(виконано самостійно)

Дані на цій сторінці оновлюються в режимі онлайн самостійно, та ретранслюють таблицю “Doctors”. Ось та сама Olivia котру я перевіряв дані попередньо.

Візуальний вигляд таблиці Doctors наступний.

Таблиця: myapp1_doctors Filter in any column

	id	FirstName	LastName	Address	Phone	Profession	Department
	Фільтр	Фільтр	Фільтр	Фільтр	Фільтр	Фільтр	Фільтр
1	1	Brian	Davis	40202 MAPLE LN GREENVILLE SC 29607-1234 USA	+15674329876	DC	Neurology
2	7	Emma	Perez	30505 BIRCH RD TINYTOWN NY 54321-6789 USA	+182943397692	DC	Neurology
3	11	Sophia	Evans	31010 PINE ST TINYTOWN NY 54321-6789 USA	+182943397697	DC	Neurology
4	19	Olivia	Carter	30606 PINE AVE MEDIUMTOWN NY 12345-6789 ...	+182943397693	DC	Gynecology
5	44	Sophia	Evans	31010 PINE ST TINYTOWN NY 54321-6789 USA	+182943397697	DC	Neurology
6	45	Olivia	Carter	30606 PINE AVE MEDIUMTOWN NY 12345-6789 ...	+182943397693	DC	Gynecology
7	46	Liam	Turner	30707 OAK ST BIGTOWN NY 98765-4321 USA	+182943397694	Medsis	Neurology
8	47	Ava	Davis	30808 ELM RD NEWTOWN NY 54321-6789 USA	+182943397695	Medsis	Gynecology
9	48	Jackson	Rodriguez	30909 CHERRY LN LARGETOWN NY 98765-4321 ...	+182943397696	DC	Cardiology
10	49	Sophia	Evans	31010 PINE ST TINYTOWN NY 54321-6789 USA	+182943397697	DC	Neurology
11	50	Aiden	Cooper	31111 OAK AVE MEDIUMTOWN NY 12345-6789 ...	+182943397698	Medsis	Neurology
12	51	Emma	Perez	31212 ELM LN BIGTOWN NY 98765-4321 USA	+182943397699	Medsis	Neurology
13	52	Olivia	Carter	31313 CHERRY RD NEWTOWN NY 54321-6789 USA	+182943397700	IN	Neurology
14	53	Liam	Turner	31414 PINE AVE LARGETOWN NY 98765-4321 USA	+182943397701	Medsis	Neurology
15	54	Ava	Davis	31515 OAK ST TINYTOWN NY 54321-6789 USA	+182943397702	Medsis	Neurology
16	55	Jackson	Rodriguez	31616 ELM RD MEDIUMTOWN NY 12345-6789 USA	+182943397703	IN	Neurology
17	56	Sophia	Evans	31717 CHERRY LN BIGTOWN NY 98765-4321 USA	+182943397704	DC	Gynecology
18	57	Aiden	Cooper	31818 PINE ST NEWTOWN NY 54321-6789 USA	+182943397705	Medsis	Neurology
19	58	Emma	Perez	31919 OAK AVE LARGETOWN NY 98765-4321 USA	+182943397706	DC	Gynecology
20	59	Olivia	Carter	32020 ELM LN TINYTOWN NY 54321-6789 USA	+182943397707	DC	Orthopedics
21	60	Liam	Turner	32121 CHERRY RD MEDIUMTOWN NY 12345-678...	+182943397708	IN	Neurology
22	61	Ava	Davis	32222 PINE AVE BIGTOWN NY 98765-4321 USA	+182943397709	Medsis	Cardiology
23	62	Jackson	Rodriguez	32323 OAK ST NEWTOWN NY 54321-6789 USA	+182943397710	IN	Orthopedics
24	63	Sophia	Evans	32424 ELM RD LARGETOWN NY 98765-4321 USA	+182943397711	Medsis	Orthopedics
25	64	Aiden	Cooper	32525 CHERRY LN TINYTOWN NY 54321-6789 USA	+182943397712	DC	Neurology
26	65	Emma	Perez	32626 PINE ST MEDIUMTOWN NY 12345-6789 USA	+182943397713	DC	Gynecology
27	66	Olivia	Carter	32727 OAK AVE BIGTOWN NY 98765-4321 USA	+182943397714	Medsis	Gynecology
28	67	Liam	Turner	32828 ELM LN NEWTOWN NY 54321-6789 USA	+182943397715	IN	Orthopedics
29	68	Ava	Davis	32929 CHERRY RD LARGETOWN NY 98765-4321 ...	+182943397716	IN	Neurology
30	69	Jackson	Rodriguez	33030 PINE AVE TINYTOWN NY 54321-6789 USA	+182943397717	DC	Orthopedics

1 - 31 з 66 Перейти до: 1

Рисунок 4.14 — Таблиця Doctor (виконано самостійно)

Вигляд заповненої таблиці зображений у рисунок 4.14. Стовбці “profession” мають абривіатури проте у відображені таблиці усе відображається корректно. Кожна абривіатура професії має свою назву назви показані нижче.

Таблиця 4.1 Таблиця скорочень у коді (виконана самостійно)

Скорочення	Profession(професії)
професор	PR
лікар	DC
медсестра	Medsis
інтерн	IN

Наступним етапом є кластеризування та стисненя бази даних. Для зменшення обсягу та поліпшення продуктивності бази даних. Стиснення нам допомагає заощаджувати місце на сервері, збільшити швидкість передачі даних по мережі та збільшити продуктивність. Кластеризація допоможе підвищити продуктивність запитів, поліпшити якості аналізу, оптимізувати базу даних.

4.2 Встановлення актуального кластирезатора для бази даних

Для встановлення більш оптимальної кластеризації для даних з метою підвищення швидкості пошуку в базі даних був проведений експеримент з використанням кластеризаторів “K-Means” та “Affinity Propagation”[1]. Після виконання експерименту можна підбити короткий підсумок, що класифікатор “K-Means” є алгоритмом котрий розбиває набір даних на кластери, основна ідея цього алгоритму полягає у мінімалізації середньоквадратичної відстані між точками даних та центроїдами кластирів.

Середньоквадратична відстань “J” котра мінімізується в процеси навчання, може бути виражена наступною формулою:

$$J = \sum_{i=1}^{n} \sum_{j=1}^{K} \left\| x_i - c_j \right\|^2 \qquad (4.1)$$

де

x_i – точка даних;

c_j – центроїд кластера.

А алгоритм класифікатора “Affinity Propagation” він використовує міру "схожості" між парами даних виділення кластерів. Замість явної вказівки числа кластерів, як у K-Means, алгоритм Affinity Propagation вибирає кількість кластерів автоматично.

На приклад: Нехай *s(i,k)* це "схожість" між точкою *i* та точкою *k.* Також вводиться значення "відповідальності" (*r(i,k)*) яке вимірює, наскільки точка *k* підходить для стати "представником" для точки *i,* та значення "доступності"*(a(i,k)),* які вимірюють, наскільки точка *i* підходить для стати "представником" для точки *k.*

Отже алгоритм виконується наступним чином:

Спершу виконується ініціалізація:

r(i,k) = 0 для всіх *i* та *k;*

a(i,k) = 0 для всіх *i* та *k.*

Наступним виконується оновлення відповідальності та доступності:

$$r(i,k) = s(i,k) - max_{k` \neq k}\{a(i,k`) + s(i,k`)\} \quad (4.2)$$

$$a(i,k) = \min\left\{0, r(k,k) + \sum_{i` \neq i, i` \neq k} max\{0, r(i`,k)\}\right\} \text{ для всіх } i \neq k$$

Далі розрахунок виконується на основі значень відповідальності та доступності.

Отже всі данні в базі даних мають текстову форму. Для кращого розуміння роботи алгоритму K-means важливо знати принцип та метод розрахунку. Перш за все це встановити відстань між точками. Для визначення близькості точок до центроїдів зазвичай використовується евклідова відстань. Для двох точок “x” та “y”. *n* - у вимірному просторі формула виглядає наступним чином:

Евклідова відстань

$$x, y = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \quad (4.3)$$

Наступним етапом є вибір найближчого центроїду. Кожна точка x з даних відноситься до найближчого до неї центроїду на основі відстані. Якщо dist (x, c_j) відстань від точки "x" до центроїду "c_j", то точку "x" буде віднесена до кластера j, $j = \text{argmin}_j dist(x, c_j)$

Перерахунок центроїдів виконується після присвоєння кожній точці кластера, центроїди перераховуються шляхом обчислення середньої для всіх точок, що належать до даного кластера. Якщо S_j - безліч точок, віднесених до кластера *j*, то новий центроїд c_j обчислюється як середнє для всіх точок у цьому кластері:

$$c_j = \frac{1}{|S_j|}\sum_{x \in S_j} x \quad (4.4)$$

Принцип алгоритму DBSCAN використовує відстань між точками для визначення відстані між двома точками x_i та x_j. де n – у вимірному просторі використовується евклідова відстань вона розраховується за наступною формулою

$$(x_i, x_j) = \sqrt{\sum_{k=1}^{n}(x_{i,k} - x_{j,k})^2} \quad (4.5)$$

Наступним етапом є виявлення ядра кластера (core points). Точка x_i вважається ядром кластера, якщо в її околі міститься не менше ніж MinPts точок. Виходячи з розрахунків вижче можна сформулювати наступний вираз

$$\left|\{x_j \middle| \text{Евклідова відстань } (x_i, x_y \leq \varepsilon)\}\right| \geq \text{MinPts} \quad (4.6)$$

Встановлення сусідніх точок відбувається за рахунок встановлення всіх точок, що знаходяться в межах відстані ε від даної точки x_i, називається сусідами цієї точки:

$$N_{(x_i)} = \{x_j \middle| \text{Евклідова відстань}(x_i, x_j) \leq \varepsilon\} \quad (4.7)$$

Розширення кластера (expanding cluster) відбувається шляхом додавання сусідів ядра кластера. Якщо точка є ядром кластера, всі її сусіди також додаються до кластера.

Наступним є порівняння двох алгоритмів стиснення, Huffman та Deflate, порівняння наведені нижче в таблиці 4.2.

Таблиця 4.2 Таблиця порівняння методів стиснення Huffman та Deflate(виконана самостійно)

Назва алгоритму	Huffman	Deflate
Особливості	Використовує коди Huffman для стиснення даних, забезпечуючи оптимальне бітове представлення для кожного символу..	Комбінує методи LZ77 та алгоритм Huffman для стиснення даних.
Переваги	Досить простий для реалізації. Зазвичай генерує менший код, ніж Deflate. Швидший процес декодування через простоту кодування.	Забезпечує кращу стискальну ефективність, особливо для текстових даних. Може генерувати більш компактний код. Може вимагати більше часу на декодування через складніші алгоритми.
Недоліки	Може бути менш ефективним для деяких типів даних. Може збільшити розмір файлу, якщо таблиця кодів займає більше простору, ніж самі дані. Деякі формати файлів можуть не підтримувати напряму коди Huffman.	Реалізація через комбінацію методів складніша. Можливі додаткові витрати обчислювальних ресурсів через складний алгоритм. У деяких випадках може збільшуватись розмір файлу через додаткові метадані та блоки.

Порівняння між алгоритмами кластеризації “K-Means” та “Affinity Propagation” [16]дала змогу зрозуміти, що класифікатор “Affinity Propagation” є більш ефективний у даній базі даних, адже цей класифікатор автоматично обирає кількість кластерів та може знаходити більш складні структури в даних, проте, це його робить більш складнішим, проте на відміну від “K-Means” котрий легший проте не завжди легко вибрати оптимальну кількість кластерів. Стосовно вибору кластерів то “Affinity Propagation”[16] автоматично встановлює кількість кластерів на основі вхідних даних на відміну від “K-Means” де потрібно попередньої вказівки кількості кластерів k , що може вимагати знання даних або використання методів, таких як "лікоть" (elbow method) для оцінки оптимального K. У базі даних присутні різні хвороби отже кластерізатор “Affinity Propagation” більш корисний, адже виявлення більш складних патернів та взаємозв'язків у текстових даних, на відміну від “K-Means” де використовується кластеризація текстових даних, котра залежить від кількості тем чи категорій.

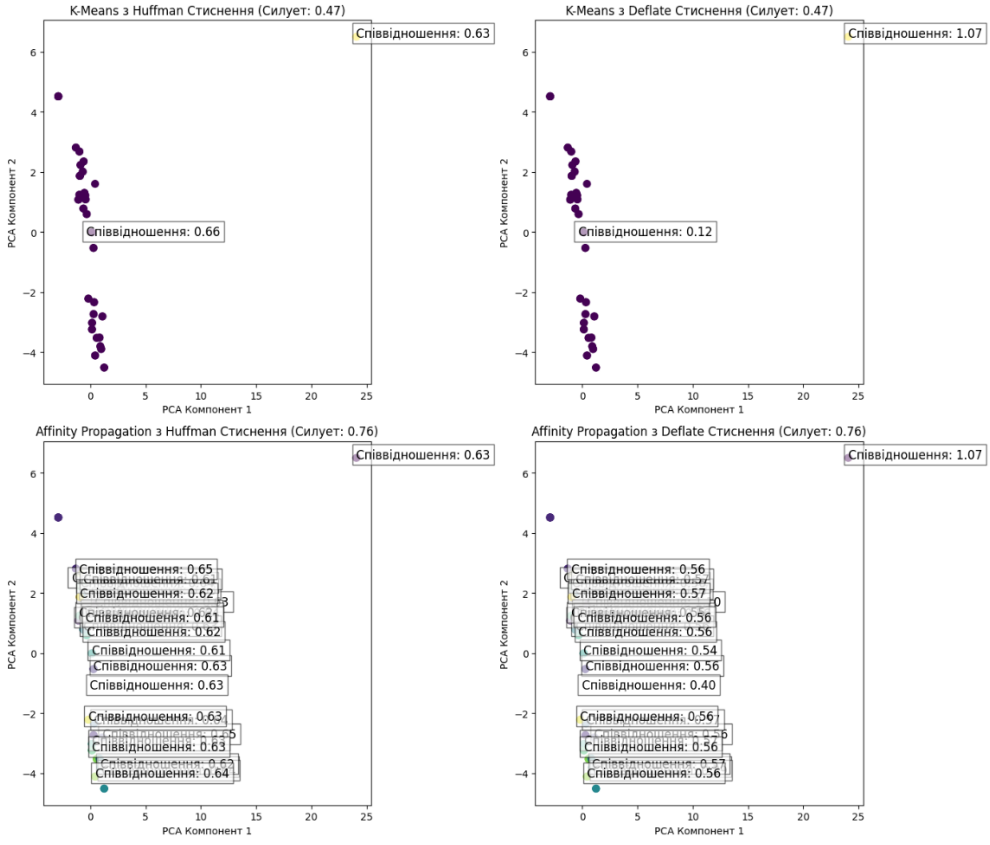


Рисунок 4.15 — Результат аналізу виконання коду перше дослідження (виконано самостійно)

На рисунку 4.15 показано результати кластеризації даних, отриманих із двох різних методів стиснення: Huffman та Deflate. Кластеризація виконана за допомогою двох алгоритмів: K-Means та Affinity Propagation.

K-Means не дуже добре працює із даними. В обох випадках (Huffman і Deflate) кластери мають погані силуети і, схоже, добре не відокремлюються один від одного. Affinity Propagation працює краще. В обох випадках (Huffman і Deflate) кластери мають вищі силуети, а точки даних добре розподілені. Huffman та K-Means мають силует 0,47, а співвідношення 0,63. Кластери не дуже розділені та близькі один до одного. Affinity Propagation має силует 0,76 в обох варіантах стиснення, а співвідношення 0,63. Дані добре розділені, і кластери мають чіткіший кордон.

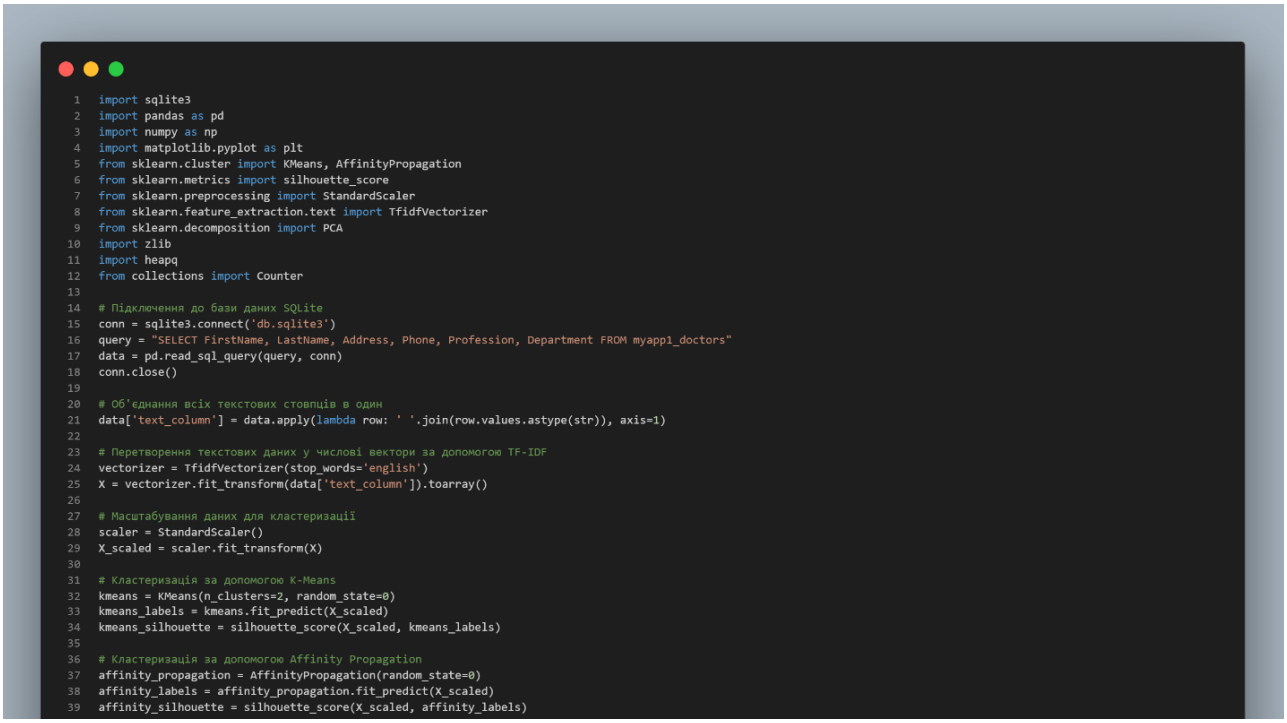
Співвідношення не є метрикою якості кластеризації. Вони вказують на те, як кластери розділяють дані, але не враховують форму і розмір кластерів.

Силует - це міра якості кластеризації, яка враховує як компактність кластерів, і поділ між ними.

Отже висновок можна зробити наступний, що Affinity Propagation[16], схоже, найбільш підходящий алгоритм кластеризації даних, отриманих з методів стиснення, як Huffman, так і Deflate[17]. Важливо зазначити, що зображення не надає інформації про вихідний набір даних, що було б важливо для подальшої інтерпретації цих результатів. Нам не відомі характеристики даних (наприклад, кількість функцій, розподіл, зв’язки між функціями), які могли б пояснити спостережувані відмінності в продуктивності кластеризації.

Щоб зрозуміти вплив методів стиснення на кластеризацію, було виконане щн одне дослідження та проведений аналіз кластеризаторів K-Means та DBSCAN, та все ті методи стиснення і були отримані наступні результати.

Весь код буде складатися з трёх частин, адже він дуже великий.



```python
import sqlite3
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans, AffinityPropagation
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import PCA
import zlib
import heapq
from collections import Counter

# Підключення до бази даних SQLite
conn = sqlite3.connect('db.sqlite3')
query = "SELECT FirstName, LastName, Address, Phone, Profession, Department FROM myapp1_doctors"
data = pd.read_sql_query(query, conn)
conn.close()

# Об'єднання всіх текстових стовпців в один
data['text_column'] = data.apply(lambda row: ' '.join(row.values.astype(str)), axis=1)

# Перетворення текстових даних у числові вектори за допомогою TF-IDF
vectorizer = TfidfVectorizer(stop_words='english')
X = vectorizer.fit_transform(data['text_column']).toarray()

# Масштабування даних для кластеризації
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Кластеризація за допомогою K-Means
kmeans = KMeans(n_clusters=2, random_state=0)
kmeans_labels = kmeans.fit_predict(X_scaled)
kmeans_silhouette = silhouette_score(X_scaled, kmeans_labels)

# Кластеризація за допомогою Affinity Propagation
affinity_propagation = AffinityPropagation(random_state=0)
affinity_labels = affinity_propagation.fit_predict(X_scaled)
affinity_silhouette = silhouette_score(X_scaled, affinity_labels)
```

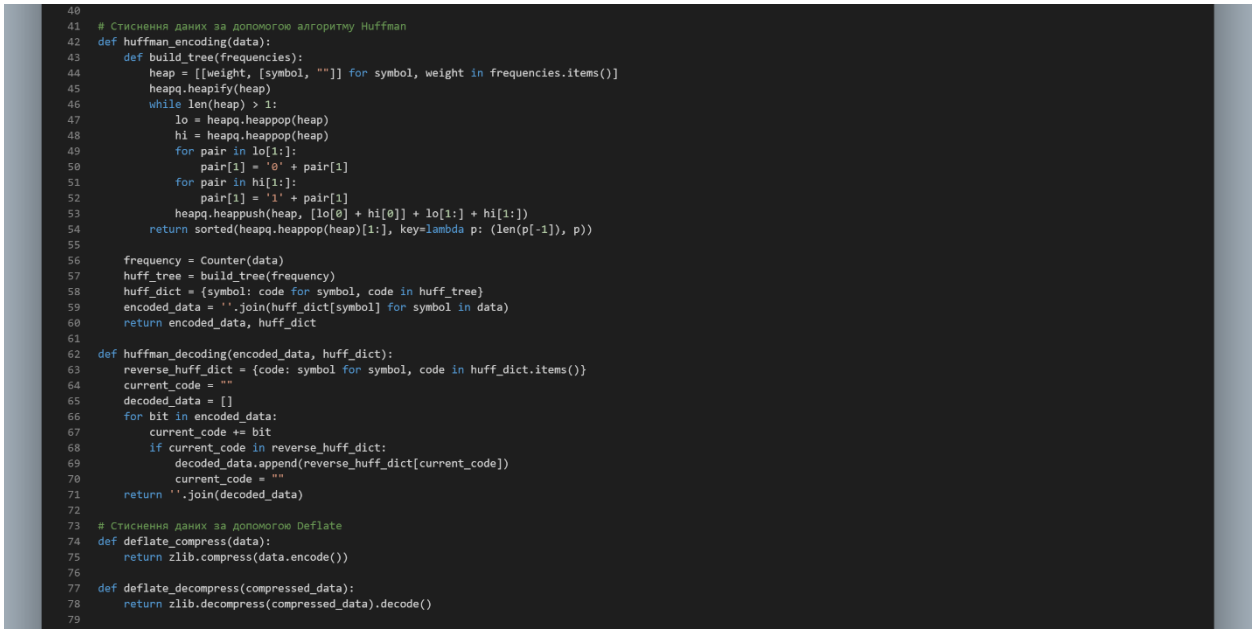
Рисунок 4.16 — Перша частина коду кластеризації та стиснення (виконано самостійно)

У першій частині коду виконуються наступні дії, підключення до бази даних та вилучення даних. Підключається до бази даних SQLite[21] з ім'ям db.sqlite3. На основі отриманих даних з таблиці myapp1_doctors, включаючи поля FirstName, LastName, Address, Phone, Profession та Department закриває з'єднання з базою даних. Попередня обробка даних, поєднує всі текстові стовпці в один стовпець text_column, щоб створити рядок, який містить усі дані для кожного запису. TF-IDF векторизація[17] перетворює текстові дані на числові за допомогою методу TF-IDF (Term Frequency-Inverse Document Frequency), виключаючи стандартні англійські стоп-слова. Це дозволяє подавати текстові дані у вигляді числових векторів. Код масштабує дані за допомогою стандартизації, щоб кожна ознака мала нульове середнє значення та одиничне стандартне відхилення. Це потрібно для коректної роботи більшості алгоритмів машинного навчання.

Кластеризація з використанням K-Means[16] застосовує алгоритм кластеризації K-Means із двома кластерами (параметр n_clusters=2). Обчислює мітки кластерів для кожного запису. Оцінює якість кластеризації за допомогою

коефіцієнта силуету “Silhouette Score”, який вимірює, наскільки добре кожна точка збігається зі своїм кластером у порівнянні з іншими кластерами.

Кластеризація з використанням Affinity Propagation застосовує алгоритм кластеризації Affinity Propagation. Він обчислює мітки кластерів для кожного запису. Оцінює якість кластеризації за допомогою коефіцієнта силуету.



```
40
41  # Стиснення даних за допомогою алгоритму Huffman
42  def huffman_encoding(data):
43      def build_tree(frequencies):
44          heap = [[weight, [symbol, ""]] for symbol, weight in frequencies.items()]
45          heapq.heapify(heap)
46          while len(heap) > 1:
47              lo = heapq.heappop(heap)
48              hi = heapq.heappop(heap)
49              for pair in lo[1:]:
50                  pair[1] = '0' + pair[1]
51              for pair in hi[1:]:
52                  pair[1] = '1' + pair[1]
53              heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
54          return sorted(heapq.heappop(heap)[1:], key=lambda p: (len(p[-1]), p))
55
56      frequency = Counter(data)
57      huff_tree = build_tree(frequency)
58      huff_dict = {symbol: code for symbol, code in huff_tree}
59      encoded_data = ''.join(huff_dict[symbol] for symbol in data)
60      return encoded_data, huff_dict
61
62  def huffman_decoding(encoded_data, huff_dict):
63      reverse_huff_dict = {code: symbol for symbol, code in huff_dict.items()}
64      current_code = ""
65      decoded_data = []
66      for bit in encoded_data:
67          current_code += bit
68          if current_code in reverse_huff_dict:
69              decoded_data.append(reverse_huff_dict[current_code])
70              current_code = ""
71      return ''.join(decoded_data)
72
73  # Стиснення даних за допомогою Deflate
74  def deflate_compress(data):
75      return zlib.compress(data.encode())
76
77  def deflate_decompress(compressed_data):
78      return zlib.decompress(compressed_data).decode()
79
```

Рисунок 4.17 — Друга частина коду кластеризації та стиснення (виконано самостійно)

Друга частина коду надає дві реалізації методів стиснення даних: алгоритм Huffman та алгоритм Deflate. Давайте докладно розглянемо, що відбувається у кожному методі.

Алгоритм Huffman використовується для втрат стиснення даних. Він створює префіксний код для кожного символу, ґрунтуючись на частоті його появи даних. Символи, які зустрічаються частіше, отримують короткі коди, а символи, що зустрічаються рідше, отримують довші коди. Функція huffman_encoding(data)[2] виконує створення частотного дерева, frequency = Counter(data) створює словник, де ключі - це символи, а значення - їх частоти даних. Heap = [[weight, [symbol, ""]] for symbol, weight in frequencies.items()][2] створює список, де кожен елемент - це

частота символу та пари (символ, його код). Heapq.heapify(heap) перетворює список на купу (heap), що дозволяє ефективно отримувати мінімальні елементи. Поки в купі більше одного елемента, виконуються такі кроки, дістаються два елементи з найменшою частотою. Коди цих елементів оновлюються, '0' додається до першого коду елемента, '1' - до другого коду.

Ці два елементи поєднуються в один з новою частотою, що дорівнює сумі їх частот, і повертаються в купу. Return sorted(heapq.heappop(heap)[1:], key=lambda p: (len(p[-1]), p)), повертає відсортований список пар (символ, код).

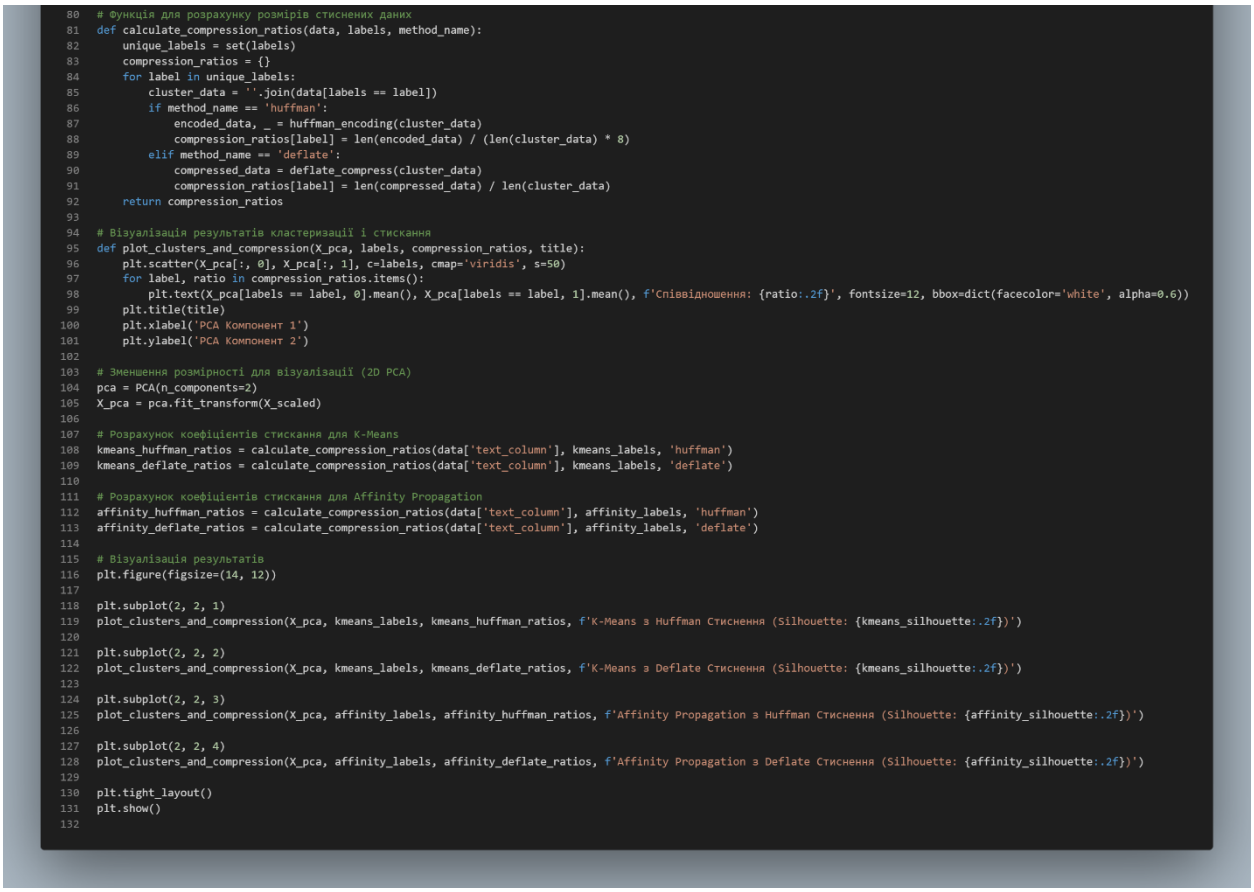
Кодування даних відбувається у частиці коду huff_dict = {symbol: code for symbol, code in huff_tree}де створює словник, де ключі – символи, а значення – їх коди. Encoded_data = ''.join(huff_dict[symbol] for symbol in data) кодує дані, замінюючи кожен символ на його код. Повернення результату, return encoded_data, huff_dict виконує повернення закодованих даних та словник кодів. Функція huffman_decoding(encoded_data, huff_dict) виконує створення зворотного словника кодів. Reverse_huff_dict = {code: symbol для symbol, code in huff_dict.items()} створює словник, де ключі - коди, а значення - символи.

Декодування даних виконується побитий прохід за закодованими даними.

Якщо поточний код знайдено у зворотному словнику, додає відповідний символ до результату та скидає поточний код. Return ''.join(decoded_data): Повертає декодовані дані.

Алгоритм Deflate – це алгоритм стиснення даних, який поєднує кодування Huffman та LZ77. У цьому випадку використовується бібліотека zlib для стиснення та розпакування даних. Функція deflate_compress(data) спершу стиснює дані завдяки return zlib.compress(data.encode()) стискає дані, попередньо перетворивши їх на байти (за допомогою .encode()).

Функція deflate_decompress(compressed_data)[2] у цієї функції займається розпакуванням даних return zlib.decompress(compressed_data).decode(), вона розпаковує та перетворює їх назад у рядок (за допомогою .decode()).



```python
# Функція для розрахунку розмірів стиснених даних
def calculate_compression_ratios(data, labels, method_name):
    unique_labels = set(labels)
    compression_ratios = {}
    for label in unique_labels:
        cluster_data = ''.join(data[labels == label])
        if method_name == 'huffman':
            encoded_data, _ = huffman_encoding(cluster_data)
            compression_ratios[label] = len(encoded_data) / (len(cluster_data) * 8)
        elif method_name == 'deflate':
            compressed_data = deflate_compress(cluster_data)
            compression_ratios[label] = len(compressed_data) / len(cluster_data)
    return compression_ratios

# Візуалізація результатів кластеризації і стискання
def plot_clusters_and_compression(X_pca, labels, compression_ratios, title):
    plt.scatter(X_pca[:, 0], X_pca[:, 1], c=labels, cmap='viridis', s=50)
    for label, ratio in compression_ratios.items():
        plt.text(X_pca[labels == label, 0].mean(), X_pca[labels == label, 1].mean(), f'Співвідношення: {ratio:.2f}', fontsize=12, bbox=dict(facecolor='white', alpha=0.6))
    plt.title(title)
    plt.xlabel('PCA Компонент 1')
    plt.ylabel('PCA Компонент 2')

# Зменшення розмірності для візуалізації (2D PCA)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Розрахунок коефіцієнтів стискання для K-Means
kmeans_huffman_ratios = calculate_compression_ratios(data['text_column'], kmeans_labels, 'huffman')
kmeans_deflate_ratios = calculate_compression_ratios(data['text_column'], kmeans_labels, 'deflate')

# Розрахунок коефіцієнтів стискання для Affinity Propagation
affinity_huffman_ratios = calculate_compression_ratios(data['text_column'], affinity_labels, 'huffman')
affinity_deflate_ratios = calculate_compression_ratios(data['text_column'], affinity_labels, 'deflate')

# Візуалізація результатів
plt.figure(figsize=(14, 12))

plt.subplot(2, 2, 1)
plot_clusters_and_compression(X_pca, kmeans_labels, kmeans_huffman_ratios, f'K-Means з Huffman Стиснення (Silhouette: {kmeans_silhouette:.2f})')

plt.subplot(2, 2, 2)
plot_clusters_and_compression(X_pca, kmeans_labels, kmeans_deflate_ratios, f'K-Means з Deflate Стиснення (Silhouette: {kmeans_silhouette:.2f})')

plt.subplot(2, 2, 3)
plot_clusters_and_compression(X_pca, affinity_labels, affinity_huffman_ratios, f'Affinity Propagation з Huffman Стиснення (Silhouette: {affinity_silhouette:.2f})')

plt.subplot(2, 2, 4)
plot_clusters_and_compression(X_pca, affinity_labels, affinity_deflate_ratios, f'Affinity Propagation з Deflate Стиснення (Silhouette: {affinity_silhouette:.2f})')

plt.tight_layout()
plt.show()
```

Рисунок 4.18 — Третя частина коду кластеризації та стиснення (виконано самостійно)

Обчислення коефіцієнтів стиснення відбувається за рахунок збору даних кластерів, дані групуються за мітками кластерів, і для кожного кластера об'єднуються в один рядок.

Стиснення даних алгоритму Huffman виконується наступним чином, спершу створюється дерево Huffman на основі частоти символів даних, і дані стискаються з використанням цього дерева.

Алгоритм стиснення Deflate, виконує наступний алгоритм спершу дані стискаються за допомогою бібліотеки zlib, де коефіцієнт стиснення визначається як відношення довжини стиснутих даних до довжини вихідних даних. Візуалізація кластерів та коефіцієнтів стиснення відбувається за рахунок зменшення розмірності даних, метод PCA використовує зменшення розмірності даних до двох вимірів, що спрощує візуалізацію.

Створюються графіки розсіювання кожної комбінації алгоритму кластеризації і методу стиснення. На графіках відображаються точки, кожна з яких є об'єктом даних, а колір точки вказує на кластер. Відображення коефіцієнтів стиснення: На кожному графіку кожного кластера відображається текстова мітка з коефіцієнтом стиснення.

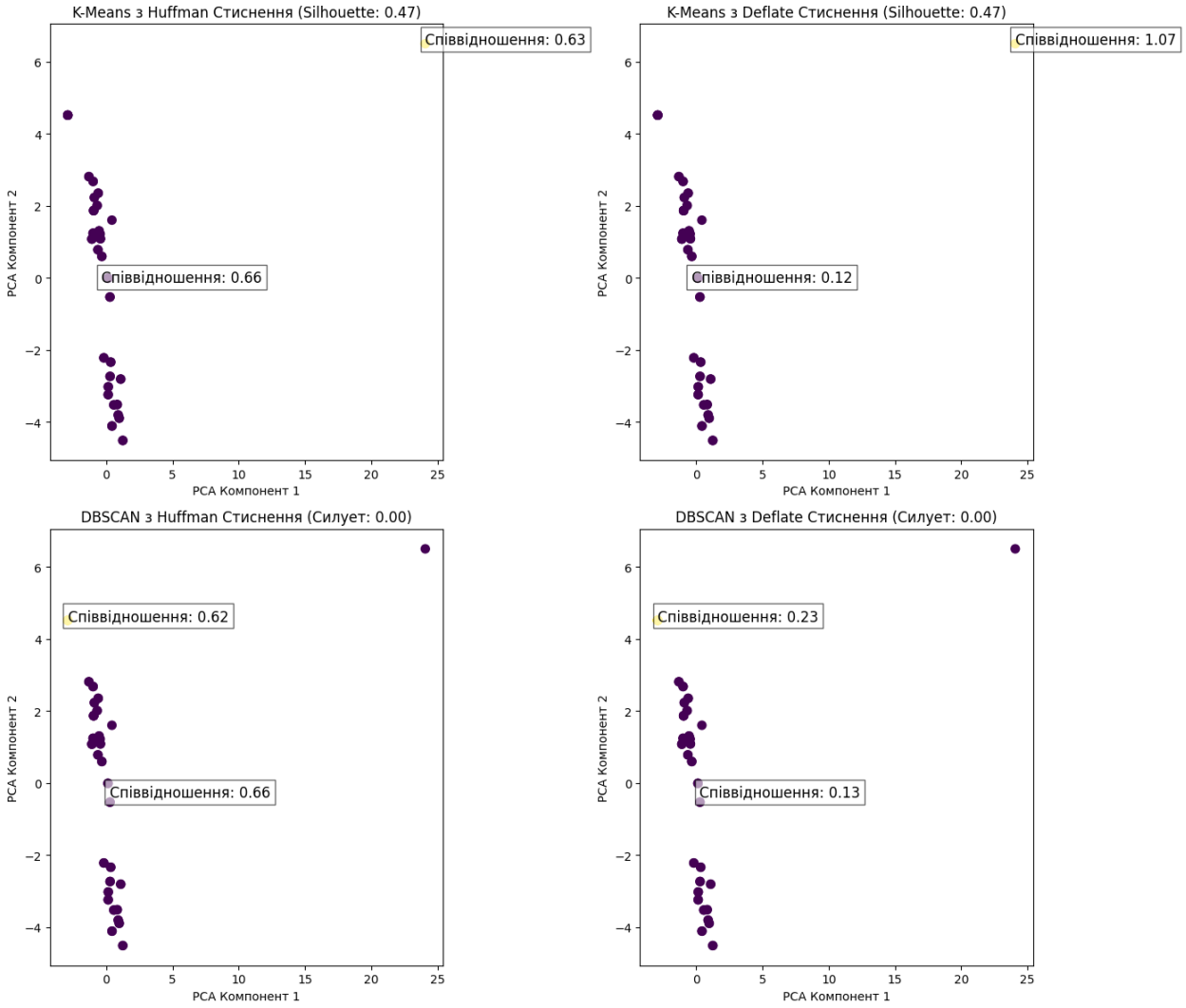


Рисунок 4.19 — Результат аналізу виконання коду друге дослідження (виконано самостійно)

На зображенні представлені результати кластеризації методом K-Means та DBSCAN для двох алгоритмів стиснення Huffman та Deflate.

K-Means та Huffman, 3 кластери, силует склав 0.47, довідково 0.63 (для 3 кластерів), 0.66 (для 2 кластерів). K-Means та Deflate стиснення виконувалося 3 кластерами, силует 0.47, довідково 1.07 (для 3 кластерів), 0.12 (для 2 кластерів).

DBSCAN та Huffman, 1 кластер, силует 0.00, довідково 0.62 (для 3 кластерів), 0.66 (для 2 кластерів). DBSCAN та Deflate стиснення виконувалося 1 кластером, силует 0.00, довідково 0.23 (для 3 кластерів), 0.13 (для 2 кластерів).

Отже можна зробити наступний висновок, що K-Means на обох наборах даних із 3 кластерами демонструє помірну якість кластеризації, але, мабуть, дані не ідеально підходять для поділу на 3 групи, проте DBSCAN в обох випадках не знайшов чітких кластерів, що може вказувати на відсутність

Висновок

Під час виконання дипломної роботи були проаналізовані всі відомі моделі та архітектурні рішення баз даних. Було обрано реляційну базу даних як дипломний проект, було проведено оптимізацію роботи бази даних шляхом кластеризації та стиснення даних. Це дозволило нам провести дослідження в двох головних напрямках: ефективності кластеризації текстових даних за допомогою алгоритмів K-Means, Affinity Propagation та DBSCAN, а також порівняння методів стиснення Huffman і Deflate для текстових даних, та встановлення оптимальних кластеризаторів.

За результатами проаналізованих даних можна зробити такі висновки, що метод K-Means не є оптимальним для кластеризації даних, отриманих з методів стиснення Huffman і Deflate. Кластери, отримані за допомогою K-Means, мають низьку силует та нечіткі межі.

Метод DBSCAN також не є ефективним для кластеризації даних, отриманих із методів стиснення Huffman та Deflate. DBSCAN не знайшов чітких кластерів даних, що може вказувати на відсутність чітких меж між групами даних. Дані, отримані з методів стиснення Huffman та Deflate, можуть мати складну структуру, яка не може бути адекватно описана традиційними методами кластеризації.

Проте метод Affinity Propagation показує вищу якість кластеризації порівняно з K-Means та DBSCAN для обох наборів даних. Для набору Huffman стиснення Affinity Propagation показує найкращу якість кластеризації серед розглянутих методів. Для набору Deflate, стиснення Affinity Propagation показує помірну якість кластеризації, але все ж таки вище, ніж у K-Means та DBSCAN.

Отже загальний висновок полягає в тому, що ефективність алгоритмів залежить від конкретного типу даних та поставлених завдань. У випадку обробки текстових даних, алгоритм Affinity Propagation для кластеризації та метод Huffman для стиснення, показали кращі результати. Ці висновки можуть бути корисними

для практичних застосувань у сфері обробки текстових даних, таких як аналіз текстів, компресія даних та інші.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Системи управління базами даних” Рагу Рамакрішнана та Йоганнеса Герке 1996р. – 599-604с.

2. Python для аналізу даних” Веса Маккіні. 2012р. – 108 – 195с.

3. Principles and best practices of scalable realtime data systems Nathan Marz with James Warren 2015р. – 27-46с.

4. Zgurovsky M.Z., Zaychenko Y.P. Big Data: Conceptual Analysis and Applications. − Springer, 2020р. – 298 с.

5. George A. Python Text Mining: Perform Text Processing, Word Embedding, Text Classification and Machine Translation / Alexandra George., 2022р. – 320 с. – (1st Edition)

6. J. Zaki M. Data Mining and Machine Learning: Fundamental Concepts and Algorithms / M. J. Zaki, W. Meira, Jr. – Cambridge: Cambridge University Press, 2020р. – 776 с. – (2nd Edition).

7. Stephen J. Wright, Benjamin Recht. Optimization for Data Analysis. – Cambridge University Press, 2022р. – 238 с.

8. Moghadas S.M., Jaberi-Douraki M. Mathematical Modelling: A Graduate Textbook. Wiley, 2018р. – 192 с.

9. Weidman S. Deep Learning from Scratch: Building with Python from First Principles / Seth Weidman., 2019р. – 235 с.

10. Database Internals: A Deep Dive into How Distributed Data Systems Work by Alex Petrov., 2023р. – 96-130 с.

11. Django 5 By Example - Fifth Edition: Build powerful and reliable Python web applications from scratch 5th ed. Edition., 2023р. – 75-97с.

12. Poguliaiev, I., Smelyakov, K., Chupryna, A., Ruban, I. Methods of Semantic Structured Search.

13. Brownlee J. Regression Metrics for Machine Learning. Machine Larning Mastery. URL: https://machinelearningmastery.com/regression-metrics-for-machine-learning (дата звернення: 30.04.2023).

14. Aggarwal C.C., Reddy C.K. Data Clustering: Algorithms and Applications. CRC Press, 2013р. – 12-48с.

15. Lukyanenko, V., Smelyakov, K., Chupryna, A., Smelyakov, S. Correlation of RFID Technology among Other Encryption Technologies.

16. Sayood K. Introduction to Data Compression. Morgan Kaufmann, 2017р. – 112-118с.

17. Silberschatz A., Korth H.F., Sudarshan S. Database System Concepts. McGraw-Hill Education, 2019р. – 38-58с.

18. Danylenko, S., Smelyakov, K., Chupryna, A. Methods of Digital-To-Analog Conversion for Reproduction of Sound Waves.

19. Lavely W. Django for Beginners: Build Websites with Python and Django. Independently published, 2018р. – 41с.

20. Date C.J. SQL and Relational Theory: How to Write Accurate SQL Code. O'Reilly Media, 2019р. – 29-41с.

21. Elmasri R., Navathe S.B. Fundamentals of Database Systems. Pearson, 2015р. – 47-64с.

22. Forcier A., Bissex D., Chun W. Python Web Development with Django. Addison-Wesley Professional, 2008р. – 4-78с.

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

12. Poguliaiev, I., Smelyakov, K., Chupryna, A., Ruban, I. Methods of Semantic Structured Search.

15. Lukyanenko, V., Smelyakov, K., Chupryna, A., Smelyakov, S. Correlation of RFID Technology among Other Encryption Technologies.

18. Danylenko, S., Smelyakov, K., Chupryna, A. Methods of Digital-To-Analog Conversion for Reproduction of Sound Waves.