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

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Кісенко Сергія Сергійовича*
	(ПІБ)
академічної групи	*121М-21-1*
	(шифр)
Спеціальності	*121 Інженерія програмного забезпечення*
	(код і назва спеціальності)
освітньої програми	*«Інженерія програмного забезпечення»*
	(назва освітньої програми)
на тему:	*Розробка програмного забезпечення для реалізації методів та алгоритмів автоматичного оновлення інформації у базі даних*

_____*С.С. Кісенко*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	*Проф. Алексєєв М.О*			

Рецензент	*Проф. Корнієнко В.І.*			

Нормоконтролер	*Проф. Лактіонов І.С.*			

Дніпро
2022

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

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

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

« » _____ 20 22 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

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

студенту *121м-21-1* *Кісенко Сергію Сергійовичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Розробка програмного забезпечення для реалізації методів та алгоритмів автоматичного оновлення інформації у базі даних*

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 31.10.2022 р. № 1200-с

2. МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес оновлення даних в БД за допомогою файлів інших програм.

Предмет досліджень – алгоритми пошуку та автоматизації роботи з файлами.

Мета НДР – підвищення якості та швидкості роботи робітників архіву.

Вихідні дані для проведення роботи – база даних клієнтів.

3. ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень визначається тим, що для цього типу підприємств даний підхід використовується в перше та орієнтований на специфіку даних.

Практична цінність результатів полягає у тому, що запропонований додаток поліпшить роботу архівного відділу, та значно пришвидшить його роботу в пошуку необхідної інформації.

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень мають бути подані у вигляді, що дозволяє побачити та оцінити безпосереднє використання нечіткої моделі даних. В результаті роботи повинен бути розроблений програмний комплекс для вирішення завдання ідентифікаційної експертизи на основі нечіткої моделі даних.

5. ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз теми та постановка завдання	
Побудова нечіткої моделі представлення даних для вирішення завдання ідентифікаційної експертизи	
Створення автоматизованої системи для вирішення завдання пошуку та автоматичного оновлення даних в БД	

6. РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект від реалізації результатів роботи очікується позитивним завдяки поліпшенню роботи з даними, що дозволить не наймати нових співробітників та швидше отримувати результати роботи від поточних.

Соціальний ефект від реалізації результатів роботи очікується позитивним завдяки удосконаленню метода методу пошуку та поновлення, що дозволяє зменшити похибку і зменшити загальний час виконання усієї роботи з даними .

7 ДОДАТКОВІ ВИМОГИ

Завдання видав _____ *Алексєєв М.О.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Кісенко С.С.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: _____ *р.*

Термін подання кваліфікаційної роботи до ЕК _____

РЕФЕРАТ

Пояснювальна записка: 54 стор., 16 рис, 4 таблиці, 2 додаток, 11 джерел.

Об'єкт дослідження: процес оновлення даних в базу даних за допомогою фалів інших програм.

Предмет дослідження: алгоритми пошуку та автоматизації роботи з файлами.

Мета роботи: підвищення якості та швидкості роботи робітників архіву.

Методи дослідження: для вирішення поставлених завдань використані методи: аналізу даних алгоритми пошуку та сортування даних.

Новизна отриманих результатів: визначається тим, що для цього типу підприємств даний підхід використовується в перше та орієнтований на специфіку даних.

Практична цінність результатів: полягає у тому, що запропонований додаток поліпшить роботу архівного відділу, та значно пришвидшить його роботу в пошуку потрібної інформації.

Область застосування: розроблений додаток був створений під потреби підприємства, але може бути також використаний при деяких дороблennях у багатьох структурах.

Значення роботи та висновки: від реалізації результатів роботи очікується позитивним завдяки поліпшенню роботи з даними, що дозволить не наймати нових співробітників та швидше отримувати результати роботи від поточних.

Прогнози щодо розвитку досліджень: покращити модульну систему, щоб додаток було можна було використовувати на інших підприємствах та покращити дизайн.

Список ключових слів: Бази даних, Індексатори, Алгоритми, Пошук, Excel файли, C#, XML.

ABSTRACT

Explanatory note: 54 pages, 16 figures, 4 tables, 2 appendix, 11 sources.

Research object: the process of updating data in the database using files of other programs.

The subject of research - search algorithms and the automation of work with files. The purpose of the work: to improve the quality and speed of work of archive workers.

Research methods - to solve the problems, the following methods were used: data analysis, data search and sorting algorithms.

The novelty of the obtained results - determined by the fact that for this company this approach is used for the first time and is focused on the specifics of the data.

The practical value of the results: is that the proposed application will improve the work of the archival department and significantly speed up their work in finding the necessary information.

Field of application. The developed application was created for the needs of the enterprise, but it can also be used with some modifications in many structures.

Value of work and conclusions. The implementation of work results is expected to be positive due to the improvement of work with data, which will allow not to hire new employees and to obtain work results more quickly from the current ones.

Forecasts regarding the development of research. improve the modular system so that the application can be used at other enterprises and improve the design.

List of keywords: Databases, Indexers, Algorithms, Search, Excel files, C#, XML.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД – база даних;

MVVM - Model-View-ViewModel;

UML – Unified Modeling Language.

XML - eXtensible Markup Language

EF Core - Entity Framework Core

ЗМІСТ

Вступ

Актуальність теми: Отримання достовірної інформації в сучасну епоху є серйозною проблемою, з якою стикаються організації. Це завдання вимагає швидкого доступу або до єдиного загального джерела інформації, або до добре організованої системи збирання даних з різних джерел. В останньому випадку проблемою є те, що для усіх проблем є рішення, яке могло б задовольнити специфічні потреби деяких компаній, тому цей проект розроблений для рішення подібної проблеми.

Об’єкт досліджень: процес оновлення даних в базу даних за допомогою фалів інших програм .

Предмет досліджень: алгоритми пошуку та автоматизації роботи з файлами.

Мета роботи: підвищення якості та швидкості роботи робітників архіву.

Мета дослідження: створення додатку, що зможе покрити ряд проблем пов’язаних з обробкою даних архіва та прискорити роботу з пошуку.

Методи дослідження: Для вирішення поставлених завдань використані методи: аналізу даних алгоритми пошуку та сортування даних

Новизна запропонованих рішень: визначається тим, що для цього підприємства даний підхід використовується в перше та орієнтований на специфіку даних.

Практичне значення: полягає у тому, що запропонований додаток поліпшить роботу архівного відділу, та значно пришвидшить їх роботу в пошуку потрібної інформації.

Особистий внесок автора: Створення додатку, якому не існує аналогів, для рішення поточної проблеми на цьому підприємстві.

Структура та обсяг дипломної роботи: робота складається з вступу, трьох розділів і висновків. Містить 54 сторінок друкованого тексту, 16 рисунків, 9 перелік використаних джерел, 2 додатків.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Методи доступу до даних та алгоритми пошуку

Методи доступу до даних таблиці поділяються на такі категорії:

1) послідовні;
2) прямі;
3) індексно-послідовні.

Метод послідовного доступу виконує послідовний перегляд усіх записів таблиці, щоб знайти те, що вам потрібно. Цей метод доступу є дуже неефективним і займає значну кількість часу для пошуку, який прямо пропорційний розміру таблиці (кількості записів). Тому його можна використовувати для відносно невеликих таблиць.

Прямий доступ вибирає необхідні записи з таблиці на основі ключа або індексу. При цьому ніякі інші записи не переглядаються. Як згадувалося раніше, значення ключа та індексу розташовані в упорядкованому форматі та містять посилання, які вказують на розташування відповідних записів у таблиці. Під час пошуку запису ви отримуєте доступ до запису безпосередньо за посиланням, а не проходите через всю таблицю.

Індексований послідовний метод доступу включає елементи методу послідовного доступу, методу прямого доступу та використовується під час отримання груп записів. Цей метод реалізується, лише якщо в полі є індекс, і його значення потрібно знайти. Його суть полягає в тому, що визначається індекс першого запису, що задовольняє задану умову і за посиланням вибирається відповідний запис із таблиці. Це прямий доступ до даних. Після обробки першого знайденого запису відбувається перехід до наступного значення індексу та вибір із таблиці запису, що відповідає цьому значенню індексу. Таким чином послідовно обходяться індекси всіх записів, що задовольняють дану умову. Це послідовний доступ.

Перевагою прямого та індексного послідовного методів є максимально висока швидкість доступу до даних. Ціною є втрата пам’яті для зберігання інформації про ключі та індекси.

Зазначений спосіб доступу реалізований СУБД і не потребує спеціального програмування. Завдання розробника – визначити правильну структуру бази даних. У цьому випадку визначте ключ та індекс. Таким чином, індексований послідовний метод доступу автоматично використовується під час отримання записів у цьому полі, оскільки для поля створено індекс. В іншому випадку використовується послідовний метод.

При виконанні операцій над таблицею використовується один із таких методів доступу до даних:

1) навігаційний;
2) реляційний.

Метод навігації доступу полягає в роботі з окремими записами таблиці. Цей метод зазвичай використовується для локальних баз даних або невеликих віддалених баз даних. Якщо потрібно обробити кілька записів, усі вони обробляються по порядку.

Методи реляційного доступу засновані на обробці груп записів одночасно, але коли потрібно обробити один запис, обробляється група з одного запису. Оскільки метод реляційного доступу заснований на запитах SQL, його також називають SQL-орієнтованим.

Цей метод доступу зосереджений на виконанні операцій із віддаленими базами даних, і хоча його також можна використовувати для локальних баз даних, він є кращим для роботи з такими базами даних.

Спосіб доступу до даних вибирається програмістом і залежить від методу доступу до бази даних, який використовується для розробки програми.

Алгоритм пошуку – це алгоритм, який розв’язує задачу пошуку. Тобто знайти інформацію, що зберігається в певній структурі даних. Структури даних можуть бути реалізовані за допомогою пов’язаних списків, масивів, дерев пошуку, хеш-таблиць або інших методів зберігання інформації. Алгоритм

пошуку рядка залежить від структури даних, у якій він реалізований. Алгоритми пошуку часто містять спеціальні команди (такі як SQL SELECT), які визначають структуру даних.

Алгоритми пошуку класифікуються на основі механізму пошуку. Лінійний пошук працює дуже повільно порівняно з двійковим, але він часто зустрічається на практиці. Алгоритм лінійного пошуку порівнює кожен елемент у структурі даних зі значенням, яке шукається. Існують також основні поняття або пошук напівінтервалів, але він працює лише для відсортованих елементів. Це один із найшвидших алгоритмів, який ітеративно порівнює пошуковий елемент із центральним елементом даної структури, ітеративно порівнюючи пошуковий елемент, якщо знайдено останній і в іншому випадку – його, потім порівняйте елементи та продовжуйте пошук у лівій чи правій частині структури даних, доки не буде знайдено правильний. У такій структурі даних кожен елемент відповідає певному ключу і дані сортуються за цими ключами, що ускладнює реалізацію алгоритмів пошуку хеш-таблиці. Оскільки алгоритм пошуку спочатку шукає ключ, а потім отримує значення за ключем, нам потрібно реалізувати хеш-функцію для пошуку елемента за ключем.

1.2 Огляд типів індексів Oracle, MySQL, PostgreSQL, MS SQL

Сімейство B-Tree індексів - це найбільш часто використовуємий тип індексів, організованих як збалансоване дерево, упорядкованих ключів. Вони підтримуються майже всіма СУБД як реляційними так і нереляційними і майже всіма типами даних.

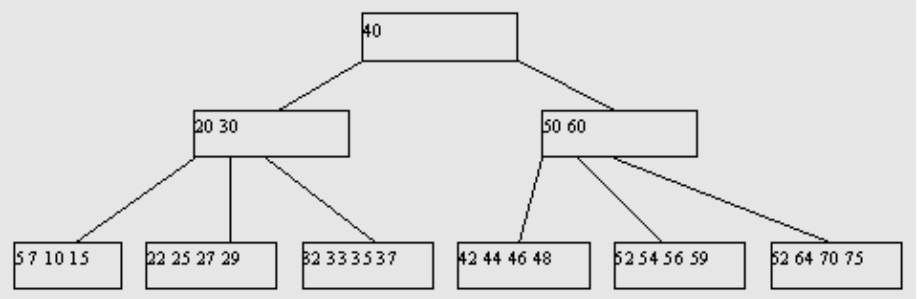

Рис. 1.1. Структура B-Tree

Слід зазначити, це те, що даний тип індексу оптимальний для множин з хорошим розподілом значень і високою потужністю (cardinality-кількість унікальних значень).

На даний момент всі дані СУБД мають просторові типи даних і функції для роботи з ними, для Oracle - це безліч типів і функцій у схемі MDSYS, для PostgreSQL - point, line, lseg, polygon, box, path, polygon, circle, MySQL - Geometry, point, linestring, polygon, multipoint, multilinestring, multipolygon, geometrycollection, MS SQL - Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon, GeometryCollection.

У схемі роботи просторових запитів зазвичай виділяють дві стадії або два ступені фільтрації. СУБД, які мають слабку просторову підтримку, відпрацьовують лише перший щабель (груба фільтрація, MySQL). Як правило, на цій стадії використовується наближене, апроксимоване уявлення об'єктів. Найпоширеніший тип апроксимації – мінімальний обмежуючий прямокутник (MBR – Minimum Bounding Rectangle) .

Для просторових типів даних існують спеціальні методи індексування на основі R-дерев (R-Tree index) і сіток (Grid-based Spatial index).

Spatial grid (просторова сітка) index – це деревоподібна структура, подібна до B-дерева, але використовується для організації доступу до просторових (Spatial) даних, тобто для індексації багатовимірної інформації, такої, наприклад, як географічні дані з двовимірними координатами (широтою та довготою). У цій структурі вузлами дерева виступають осередки простору. Наприклад, для двомірного простору: спочатку вся батьківська площа буде розбита на сітку суворо певного дозволу, потім кожен осередок сітки, в якій кількість об'єктів перевищує встановлений максимум об'єктів в осередку, буде розбита на підсітку наступного рівня. Цей процес буде продовжуватися доти, доки не буде досягнуто максимум вкладеності (якщо встановлено), або доки все не буде розділено до осередків, що не перевищують максимум об'єктів.

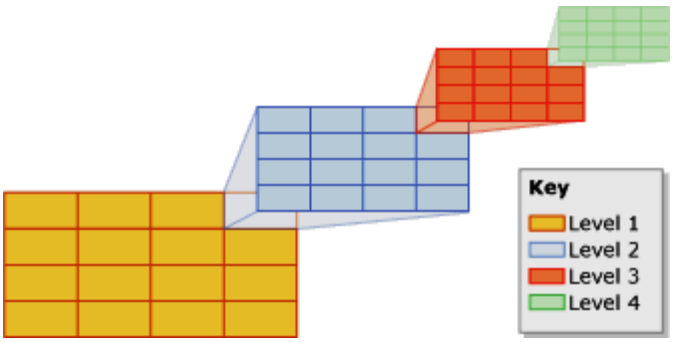

Рис. 1.2. Структура Spatial grid

У разі тривимірного чи багатовимірного простору це будуть прямокутні паралелепіпеди (кубоїди) або паралелотопи.

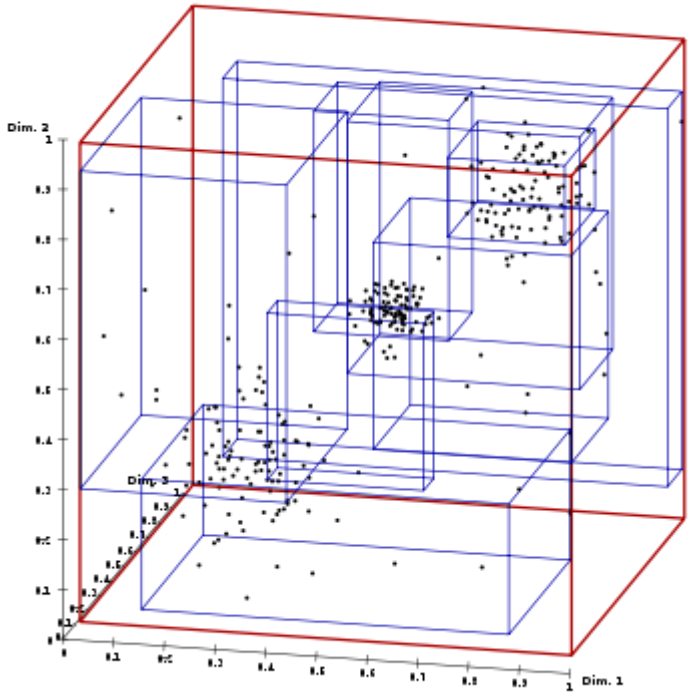

Рис. 1.3. Структура Spatial grid у багатовимірного просторі

Quadtree - це підвид Grid-based Spatial index, в якому в батьківському осередку завжди 4 нащадки і роздільна здатність сітки варіюється в залежності від характеру або складності даних.

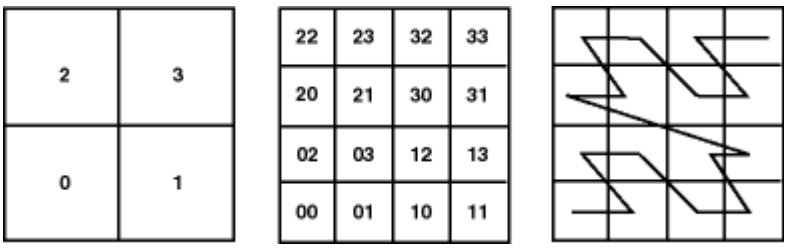

Рис. 1.4. Структура Quadtree

R-Tree (Regions Tree) – це також деревоподібна структура даних подібна до Spatial Grid, запропонована в 1984 році Антоніном Гуттманом. Ця структура даних теж розбиває простір безліч ієрархічно вкладених осередків, але які, на відміну Spatial Grid, нічого не винні повністю покривати батьківську комірку і можуть перетинатися.

Для розщеплення переповнених вершин можуть застосовуватися різні алгоритми, що породжує поділ R-дерев на підтипи: з квадратичною та лінійною складністю (Гуттман, звичайно, описав і з експоненційною складністю – Exhaustive Search, але він, природно, ніде не використовується).

Квадратичний підтип полягає в розбитті на два прямокутники з мінімальною площею, що покривають усі об'єкти. Лінійний – у розбиття за максимальною віддаленістю.

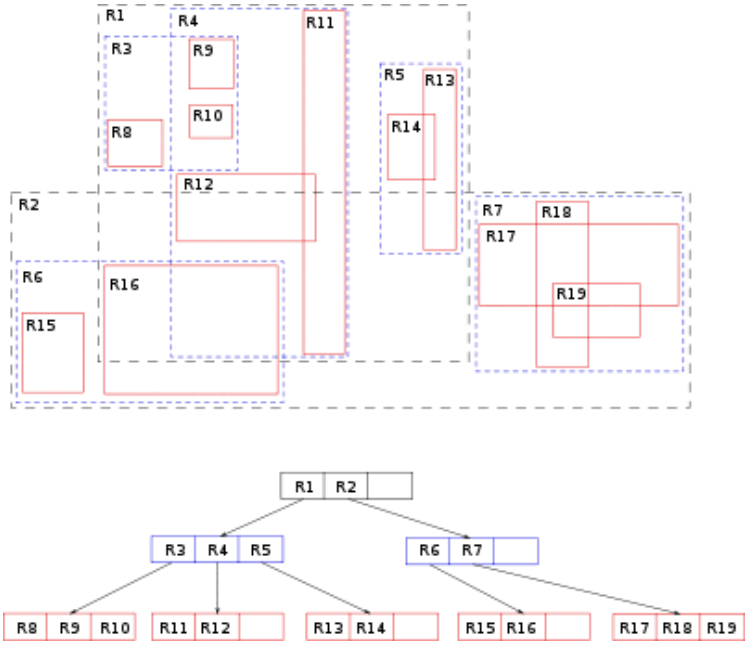

Рис. 1.5. Структура R-Tree

Hash-індекси були запропоновані Артуром Фуллером, і припускають зберігання не самих значень, а їх хешей, завдяки чому зменшується розмір (а відповідно і збільшується швидкість їх обробки) індексів великих полів. Таким чином, при запитах з використанням HASH-індексів, порівнюватися будуть не шукане значення поля, а хеш від шуканого значення з хешами полів.

Через нелінійність хеш-функцій цей індекс неможна сортувати за значенням, що призводить до неможливості використання в порівняннях більше/менше і «is null». Крім того, так як хеші не унікальні, то для хешей, що збігаються, застосовуються методи дозволу колізій.

Bitmap index – метод бітових індексів полягає у створенні окремих бітових карт (послідовність 0 і 1) для кожного можливого значення стовпця, де кожному біту відповідає рядок із значенням, що індексується, а його значення дорівнює 1 означає, що запис, відповідна позиції біта містить індексоване значення для даного стовпця чи властивості.

Поле в таблице(bin)	Ключ reverse-индекса(bin)
00000001	10000000
…	…
00001001	10010000
00001010	01010000
00001011	11010000

Таб. 1.1. Структура Bitmap index

Як бачите, значення в індексі змінюється набагато більше, ніж саме значення в таблиці і тому в структурі b-tree вони потраплять у різні блоки.

Інвертований індекс - це повнотекстовий індекс, що зберігає для кожного лексеми ключів відсортований список адрес записів таблиці, які містять даний ключ.

1	Мама мила раму
2	Тато мив раму
3	Тато мив машину
4	Мама відполірувала машину

Таб. 1.2. Структура інвертованого індексу

У спрощеному вигляді це виглядатиме так:

Мама	1,4
Мила	1
Раму	1,2
Папа	2,3
відполірувала	4
Машину	3,4

Таб. 1.3. Структура Інвертованого індексу (2)

Partial index - це індекс, побудований на частині таблиці, що задовольняє певну умову самого індексу. Цей індекс створено для зменшення розміру індексу.

Function-based index. Самим гнучким типом індексів є функціональні індекси, тобто індекси, ключі яких зберігають результат функцій користувача. Функціональні індекси часто будуються для полів, значення яких проходять попередню обробку перед порівнянням у команді SQL. Наприклад, при порівнянні рядкових даних без урахування регістру символів часто використовується функція UPPER. Створення функціонального індексу з функцією UPPER покращує ефективність таких порівнянь.

Крім того, функціональний індекс може допомогти реалізувати будь-який інший відсутній тип індексів даної СУБД (крім, мабуть, бітового індексу, наприклад, Hash для Oracle)

	MySQL	PostgreSQL	MS SQL	Oracle
B-Tree index	Є	Є	Є	Є
Просторові індекси, що підтримуються (Spatial indexes)	R-Tree з квадратичним розбиттям	Rtree_GiST (використовується лінійне розбиття)	4-х рівневий Grid-based spatial index (окремі для географічних та геодезичних даних)	R-Tree с квадратичним розбиттям; Quadtree
Hash index	Лише у таблицях типу Memory	Є	немає	немає
Bitmap index	немає	Є	немає	Є
Reverse index	немає	немає	немає	Є
Inverted index	Є	Є	Є	Є
Partial index	немає	Є	Є	немає
Function based index	немає	Є	Є	Є

Таб. 1.4. Зведена таблиця типів індексів

Варто згадати, що PostgreSQL GiST дозволяє створити для будь-якого власного типу даних індекс заснований на R-Tree. Для цього необхідно продати всі 7 функцій механізму R-Tree.

1.3. Огляд збалансованих дерев

Red-black tree, RB tree. У цій структурі баланс досягається за рахунок підтримки розмальовки вершин у два кольори (червоний і чорний, як видно з назви), що підпорядковується наступним правилам:

1) Червона вершина не може бути сином червоної вершини.
2) Чорна глибина будь-якого листа однакова (чорною глибиною називають кількість чорних вершин по дорозі з кореня).
3) Корінь дерева чорний. Тут ми дещо змінюємо визначення аркуша і називаємо так спеціальні null-вершини, які заміняють відсутніх синів. Вважатимемо такі вершини чорними.

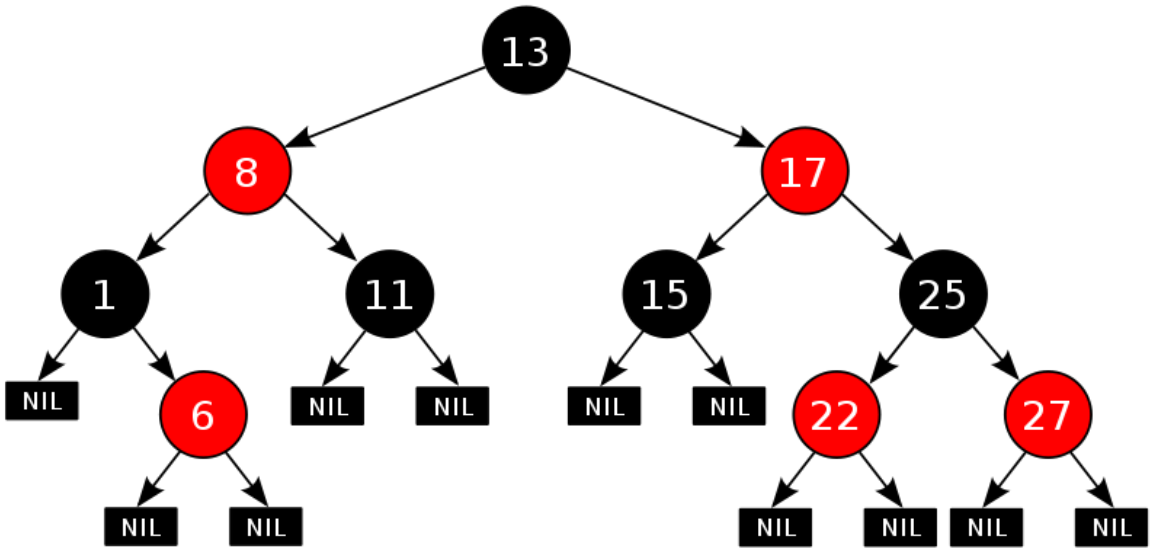

Рис. 1.6. Red-black tree

Візьмемо найглибший лист. Нехай вона знаходиться на глибині h. Через правило 1, як мінімум половина вершин на шляху кореня буде чорними, тобто чорна висота дерева буде не менше h/2. Можна показати, що в такому дереві буде не менше 2^(h/2)-1 чорних вершин (оскільки у кожної чорної вершини з чорною глибиною k, якщо вона не лист, має бути як мінімум два нащадки з чорною глибиною k+1). Тоді 2(h/2)-1 <= n або h <= 2*log2(n+1).

Усі основні операції з червоно-чорним деревом можна реалізувати за O(h), тобто O(log n) за доведеним вище. Класична реалізація заснована на аналізі великої кількості випадків і досить складна для сприйняття. Існують простіші та зрозуміліші варіанти, наприклад у статті Кріса Окасакі. На жаль, у ній описано

лише операцію вставки в дерево. Простота, порівняно з класичною реалізацією, виходить за рахунок орієнтації на зрозумілість, а не на оптимізацію кількості елементарних модифікацій дерева (обертань).

Для реалізації цього виду збалансованих дерев потрібно в кожній вершині зберігати додатково 1 біт інформації (колір). Іноді це викликає великий overhead через вирівнювання. У таких випадках переважно використовувати структури без додаткових вимог до пам'яті.

Червоно-чорні дерева широко використовуються - реалізація set/map в стандартних бібліотеках, різні застосування в ядрі Linux (для організації черг запитів, ext3 etc.), ймовірно в багатьох інших системах для аналогічних потреб.

Червоно-чорні дерева тісно пов'язані з B-деревами. Можна сміливо сказати, що вони ідентичні B-деревам порядку 4 (чи 2-3-4 деревам). Докладніше про це можна прочитати у статті на вікіпедії або у книзі «Алгоритми: побудова та аналіз», згаданій у минулій статті.

AA-дерево. Модифікація червоно-чорного дерева, де накладається додаткове обмеження: червона вершина може бути лише правим сином. Якщо червоно-чорне дерево ізоморфне 2-3-4 дереву, то AA-дерево ізоморфне 2-3 дереву.

Через додаткове обмеження операції реалізуються простіше ніж у червоно-чорного дерева (за рахунок зменшення кількості випадків, що розбираються). Оцінка на висоту дерев залишається незмінною, 2*log2(n). Ефективність за часом вони приблизно однакова, але оскільки у реалізації замість кольору зазвичай зберігають іншу характеристику («рівень» вершини), overhead по пам'яті досягає байта.

АВЛ-дерево. Придумали його двоє радянських математиків: Г.М. Адельсон-Вельського та Є.М. Ландіса.

Накладає на дерево наступне обмеження: у будь-якої вершини висоти лівого і правого піддерев'я повинні відрізнятися не більше ніж на 1. Легко довести по індукції, що дерево з висотою h повинно містити як мінімум F_h вершин, де F_i - i-е число Фібоначчі. Оскільки F_i ~ phi^n (phi=(sqrt(5)+1)/2 —

золотий переріз), висота дерева з n вершинами не може перевищити log2(n)/log2(phi) ~ 1.44*log2(n)

Реалізація, як і в червоно-чорного дерева, заснована на розборі випадків і досить складна для розуміння і має складність O(log(n)) на всі основні операції. Для роботи необхідно зберігати в кожній вершині різницю між висотами лівого та правого піддерев'я. Так як вона не перевищує 1, достатньо використовувати 2 біти на вершину.

Детальний опис можна знайти в книзі Н. Вірта «Алгоритми + структури даних = програми» або в книзі А. Шеня «Програмування: теореми та завдання»

Декартове дерево.

Якщо малювати дерево на площині, ключ буде відповідати x-координаті вершини (за рахунок упорядкованості). Тоді можна ввести і y-координату (назвемо її висотою), яка матиме наступну властивість: висота вершини більша за висоту дітей (таку ж властивість мають значення в іншій структурі даних на основі двійкових дерев — купі (heap).)

Виявляється, якщо висоти вибирати випадковим чином, висота дерева, що задовольняє властивості купи, найбільш ймовірно буде O(log(n)). Численні експерименти показують, що висота виходить приблизно 3*log(n).

Реалізація операцій проста та логічна, за рахунок цього структура дуже улюблена у спортивному програмуванні. За результатами тестування, визнана найбільш ефективною за годиною (серед червоно-чорних, АА та АВЛ — дерев, а також skip-list'ів (структура, що не є двійковим деревом, але з аналогічною областю застосування) та radix-дерев). На жаль, має досить великий overhead по пам'яті (2-4 байти на вершину, на зберігання висоти) і неприйнятна там, де потрібна гарантована продуктивність (наприклад, у ядрі ОС).

Splay-дерево. Ця структура даних дуже відрізняється від усіх перелічених раніше. Справа в тому, що вона не накладає жодних обмежень на структуру дерева. Більше того, у процесі роботи дерево може виявитися повністю розбалансованим.

Основа splay-дерева – операція splay. Вона знаходить потрібну вершину (або найближчу до неї за відсутності) і «витягує» її в корінь особливою послідовністю елементарних обертань (локальна операція над деревом, що зберігає властивість порядку, але змінює структуру). Через неї можна легко висловити всі основні операції з деревом. Послідовність операцій у splay підібрана так, щоб дерево працювало швидко.

Знаючи магію операції splay, ці дерева реалізуються не легко, а дуже легко, тому вони також дуже популярні у ACM ICPC, Topcoder etc.

Зрозуміло, що в такому дереві не можна гарантувати складність операцій O(log(n)). Натомість, гарантується амортизована складність операції O(log(n)), тобто будь-яка послідовність з m операцій з деревом розміру n працює за O((n+m)*log(n)). Більше того, splay-дерево має деякі магічні властивості, за рахунок якого воно на практиці може виявитися набагато ефективнішим за інші варіанти. Наприклад, вершини, до яких зверталися нещодавно, виявляються ближчими до кореня і доступ до них прискорюється. Більш того, доведено, що якщо ймовірності звернення до елементів фіксовані, то splay-дерево працюватиме асимптотично не повільніше за будь-яку іншу реалізацію бінарних дерев. Ще одна перевага в тому, що немає overhead по пам'яті, тому що не потрібно зберігати ніякої додаткової інформації.

На відміну від інших варіантів, операція пошуку у дереві модифікує саме дерево, тому у разі рівномірного звернення до елементів splay-дерево працюватиме повільніше. Однак на практиці воно часто дає відчутний приріст продуктивності. Тести це підтверджують - у тестах, отриманих на основі Firefox'a, VMWare та Squid'a, splay-дерево показує приріст продуктивності в 1.5-2 рази в порівнянні з червоно-чорними та АВЛ-деревами. У той же час, на синтетичних тестах splay-дерева працюють у 1.5 рази повільніше. На жаль, через відсутність гарантій на продуктивність окремих операцій, splay-дерева неприйнятні в realtime-системах (наприклад в ядрі ОС, garbage-collector'ax), а також у бібліотеках загального призначення.

Scapegoat-дерево. Це дерево схоже на попереднє тим, що воно не має overhead по пам'яті. Однак це дерево є повністю збалансованим. Більше того, коефіцієнт 0 < alpha < 0.5 "жорсткості" дерева можна задавати довільно і висота дерева буде обмежена зверху значенням k * log (n) + 1, де k = log2 (1 / alpha). На жаль, операції модифікації будуть амортизованими, як і в минулого дерева.

Коефіцієнт жорсткості сильно впливає на баланс продуктивності: чим жорсткіше дерево, тим менше у нього буде висота і тим швидше працюватиме пошук, але тим складніше підтримуватиме порядок в операціях модифікації. Наприклад, оскільки АВЛ-дерево «жорсткіше» червоно-чорного, пошук у ньому працює швидше, а модифікація повільніша. Якщо ж скористатися scapegoat-деревом, баланс між цими операціями можна вибирати в залежності від специфіки застосування дерева.

1.4 Огляд алгоритмів сортування

Сортування бульбашкою (Bubble Sort)

Один з найбільш відомих алгоритмів сортування, що часто запитуються на співбесідах. Працює за квадратичний час O(n*n). Принцип роботи досить простий: послідовно порівнюються два елементи, що стоять поруч, і якщо лівий більше правого, то вони змінюються місцями, після чого порівнюються наступні елементи. І так повторюється доти, доки всі елементи не будуть упорядковані.

Шейкерне (коктейльне) сортування (Cocktail Sort).

Модифікований та трохи покращений алгоритм бульбашкового сортування, при якому обмін виконується у двох напрямках – найбільші елементи переміщуються у праву сторону, а під час зворотного руху найменші рухаються у ліву сторону. Виконується за квадратичну годину O(n*n).

Сортування вставками (Insertion Sort)

Сортування вставками – досить простий у реалізації і зрозумілий для розуміння алгоритм, який чудово працює на частково впорядкованих послідовностях і коли колекція, що сортується, послідовно заповнюється

елементами. Працює, як і раніше розглянуті алгоритми, за квадратичний час O(n*n). Елементи послідовно додаються у відсортовану позицію у потрібне місце, доки вся колекція не буде відсортована.

Сортування Шелла (Shell Sort)

Цей алгоритм є вдосконаленою реалізацією попереднього сортування вставками. Ідея полягає в тому, щоб сортувати елементи, що стоять на деякій відстані один від одного, що в результаті дасть частково відсортовану послідовність, яку можна буде легко та швидко відсортувати вставками сортуванням. Незважаючи на те, що в гіршому випадку сортування відпрацьовує за квадратичний час O(n*n), цей алгоритм зазвичай показує значно кращий лінійно-логарифмічний результат O(n*log n).

Сортування вибором (Selection Sort)

Напевно, найпростіший з погляду розуміння алгоритм сортування – просто послідовно вибирай найбільший елемент із усієї сортованої послідовності. Але, як це часто буває, що простіше ідея та реалізація, то гірше працює. Даний алгоритм працює за квадратичний час O(n*n), що, природно, не дуже добре.

Сортування деревом (Tree Sort)

Даний алгоритм заснований на структурі даних двійкове дерево пошуку (BST), за що отримав своє ім'я. Принцип досить простий – побудувати дерево та виконати його індексний обхід.

Незважаючи на те, що в гіршому випадку, коли дерева вироджуються у зв'язковий список, час падає до квадратичного O(n*n), при нормальному балансуванні алгоритм працює за лінійно-логарифмічний час O(n*log n).

Пірамідальне сортування (Heapsort)

Сортування купою є одним з оптимальних алгоритмів сортування, що часто використовуються на практиці. Виконується він за лінійно-квадратичний час O(n*log n) і є досить стійким. Принцип його роботи, як і в попереднього алгоритму, пов'язані з структурою даних – Двійкова купа (heap).

Реалізація сортування досить проста - потрібно просто побудувати двійкову купу з мінімальним/максимальним елементом в корені і витягувати з неї кореневий елемент доти, доки в ньому будуть елементи. За рахунок балансування купи докорінно завжди буде наступний по черзі елемент.

Сортування злиттям (merge sort)

Сортування злиттям постійно бореться за пальму першості за швидкістю з наступним алгоритмом і навіть виграє за рахунок більшої стійкості. Працює очікувано за лінійно-логарифмічний O(n*log n) час і застосовує принцип "поділяй і володарюй". Вся послідовність, що сортується, розбивається навпіл на групи доти, поки в кожній з них не буде по два елементи. Потім ця маленька група впорядковується та зливається з іншою такою ж групою. При злитті в результуючу колекцію поміщаються по порядку елементи двох вихідних груп. І так доти, доки не залишиться одна відсортована послідовність.

Швидке сортування (quicksort)

Дане сортування не дарма отримало своє ім'я – найчастіше воно дозволяє виконати сортування швидше за будь-який інший алгоритм. Саме швидке сортування найчастіше застосовується практично. Працює за лінійно-логарифмічний час O(n * log n). І це був би ідеальний алгоритм, якби не одне «але»: у поодиноких випадках він може деградувати до сортування бульбашкою з квадратичним часом O(n*n). Принцип роботи алгоритму пов'язані з вибором опорного елемента, щодо якого виконується сортування. Умовно колекція ділиться навпіл щодо нього, і все, що менше за опорний елемент, переміщається вліво від нього, все, що більше – вправо. Все працює чудово, коли значення опорного елемента близько до середини, але якщо він постійно виявляється максимальним або мінімальним, то результати виявляються плачевними.

РОЗДІЛ 2
ТЕХНОЛОГІЇ ТА ПРОГРАМИ ЯКІ БУЛИ ЗАДІЯНІ В РОЗРОБЦІ

2.1 Microsoft Visual Studio 2022

Microsoft Visual Studio - лінійка продуктів компанії Microsoft, що включають інтегроване середовище розробки програмного забезпечення та низку інших інструментів. Дані продукти дозволяють розробляти як консольні програми, так і ігри та програми з графічним інтерфейсом, у тому числі з підтримкою технології Windows Forms, UWP а також веб-сайти, веб-додатки, веб-служби як в рідному, так і в керованому коді всіх платформ, підтримуваних Windows, Windows Mobile, Windows CE, .NET Framework, .NET Core, .NET, MAUI, Xbox, Windows Phone .NET Compact Framework та Silverlight. Після покупки компанії Xamarin корпорацією Microsoft з'явилася можливість розробки IOS та Android програм.

Visual Studio включає редактор вихідного коду з підтримкою технології IntelliSense і можливістю найпростішого рефакторингу коду. Вбудований налагоджувач може працювати як відладчик рівня вихідного коду, так і відладчик машинного рівня. Інші вбудовані інструменти включають редактор форм для спрощення створення графічного інтерфейсу програми, веб-редактор, дизайнер класів і дизайнер схеми бази даних. Visual Studio дозволяє створювати та підключати сторонні доповнення (плагіни) для розширення функціональності практично на кожному рівні, включаючи додавання підтримки систем контролю версій вихідного коду (як, наприклад, Subversion та Visual SourceSafe), додавання нових наборів інструментів (наприклад, для редагування та візуального проектування) коду предметно-орієнтованими мовами програмування) або інструментами для інших аспектів процесу розробки програмного забезпечення (наприклад, клієнт Team Explorer для роботи з Team Foundation Server).

2.2 SQLite

SQLite представляє бібліотеку, яка написана мовою C (ANSI-C) і яка реалізує двигун реляційних баз даних. На сьогоднішній день SQLite, можливо, система баз даних, що використовується. Так її бд можна знайти в кожному пристрої на Android, iOS, Mac, Windows 10/11, її використовуються більшість поширених браузерів - Firefox, Chrome, Safari і т.д.

На відміну від інших систем баз даних, як MS SQL Server, MySQL, Postgres і т.д., SQLite не потребує сервер бази даних. SQLite представляє вбудований двигун бази даних, який безпосередньо звертається до файлу бази даних на диску. Для роботи з базами даних нам не потрібно явно встановлювати або якось конфігурувати SQLite.

SQLite має повноцінну підтримку більшості можливостей, які мають інші реляційні СУБД - таблиці, індекси, тригери, уявлення.

Для створення запитів до бази даних SQLite застосовує мову SQL (точніше свою реалізацію), яка в цілому схожа на реалізації та діалекти SQL, які застосовуються в інших реляційних СУБД.

Формат файлу бази даних є кросплатформним - можна створити і працювати з файлом бази даних на одному пристрої з однією операційною системою, а потім спокійно скопіювати його на інший пристрій з іншої ОС.

Що стосується розробки програм більшість поширених і популярних мов програмування, таких як Python, C#, Java, і т.д., мають підтримку для SQLite, що дозволяє використовувати цю СУБД в різних сценаріях і різних типах додатків.

2.3WPF

Технологія WPF (Windows Presentation Foundation) є частиною екосистеми платформи .NET і є підсистемою для побудови графічних інтерфейсів.

Якщо при створенні традиційних програм на основі WinForms за малювання елементів керування та графіки відповідали такі частини ОС Windows, як User32 та GDI+, то програми WPF засновані на DirectX. У цьому полягає ключова особливість рендерингу графіки у WPF: використовуючи WPF, значна частина роботи з відтворення графіки, як найпростіших кнопочок, так і складних 3D-моделей, лягатиме на графічний процесор на відеокарті, що також дозволяє скористатися апаратним прискоренням графіки.

Однією з важливих особливостей є використання мови декларативної розмітки інтерфейсу XAML, заснованого на XML: ви можете створювати насичений графічний інтерфейс, використовуючи або декларативне оголошення інтерфейсу, або код керованих мов C#, VB.NET і F#, або поєднувати і те, і інше.

Перша версія - WPF 3.0 вийшла разом із .NET Framework 3.0 та операційною системою Windows Vista у 2006 році. І з того часу платформа WPF є частиною екосистеми .NET і розвивається разом із фреймворком .NET. Наприклад, на сьогоднішній день останньою версією фреймворку .NET є .NET 7 і WPF повністю підтримується цією версією фреймворку.

Переваги WPF:

1) Використання традиційних мов .NET-платформи - C#, F# та VB.NET для створення логіки програми

2) Можливість декларативного визначення графічного інтерфейсу за допомогою спеціальної мови розмітки XAML, що базується на xml і представляє альтернативу програмному створенню графіки та елементів керування, а також можливість комбінувати XAML та C#/VB.NET

3) Незалежність від роздільної здатності екрана: оскільки у WPF всі елементи вимірюються в незалежних від пристрою одиницях, програми на WPF легко масштабуються під різні екрани з різною роздільною здатністю.

4) Нові можливості, яких складно було досягти WinForms, наприклад, створення тривимірних моделей, прив'язка даних, використання таких елементів, як стилі, шаблони, теми та ін.

5) Хороша взаємодія з WinForms, завдяки чому, наприклад, у додатках WPF можна використовувати традиційні елементи керування WinForms.

6) Багаті можливості створення різних додатків: це і мультимедіа, і двомірна і тривимірна графіка, і багатий набір вбудованих елементів управління, а також можливість самим створювати нові елементи, створення анімацій, прив'язка даних, стилі, шаблони, теми та багато іншого

7) Апаратне прискорення графіки - незалежно від того, чи працюєте ви з 2D або 3D, графікою або текстом, всі компоненти програми транслюються в об'єкти, зрозумілі Direct3D, а потім візуалізуються за допомогою процесора на відеокарті, що підвищує продуктивність, робить графіку більш плавною.

8) Створення програм під безліч ОС сімейства Windows

У той же час WPF має певні обмеження. Незважаючи на підтримку тривимірної візуалізації, для створення програм з великою кількістю тривимірних зображень, перш за все ігор, краще використовувати інші засоби - DirectX або спеціальні фреймворки, такі як Monogame або Unity.

Також варто враховувати, що в порівнянні з додатками на Windows Forms обсяг програм на WPF та споживання ними пам'яті в процесі роботи в середньому дещо вищий. Але це з лишком компенсується ширшими графічними можливостями та підвищеною продуктивністю при малюванні графіки.

Крім того, незважаючи на те, що WPF працює поверх платформи .NET 5/6/7, але в силу природи WPF і залежності від компонентів Windows, на даний момент створювати програми на WPF можна тільки під ОС Windows.

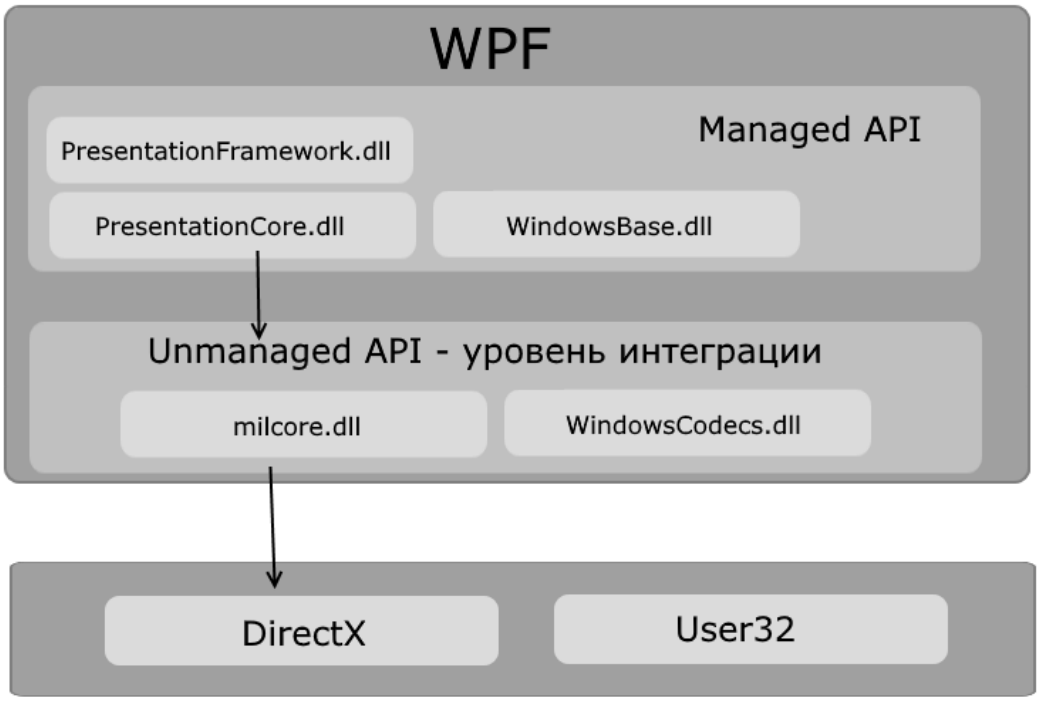

Рис. 2.1. архітектура WPF

Як видно на схемі, WPF розбивається на два рівні: managed API та unmanaged API (рівень інтеграції з DirectX). Managed API (керований API-інтерфейс) містить код, який виконується під управлінням загальномовного середовища виконання .NET - Common Language Runtime. Цей API описує основний функціонал платформи WPF і складається з наступних компонентів:

1) PresentationFramework.dll: містить усі основні реалізації компонентів та елементів керування, які можна використовувати при побудові графічного інтерфейсу

2) PresentationCore.dll: містить усі базові типи для більшості класів з PresentationFramework.dll

3) WindowsBase.dll: містить низку допоміжних класів, які застосовуються у WPF, але можуть також використовуватися і поза цією платформою

4) Unmanaged API використовується для інтеграції вищого рівня з DirectX:

5) milcore.dll: власне забезпечує інтеграцію компонентів WPF із DirectX. Цей компонент написаний на некерованому коді (C/C++) взаємодії з DirectX.

6) WindowsCodecs.dll: бібліотека, яка надає низькорівневу підтримку для зображень у WPF

Ще нижче власне знаходяться компоненти операційної системи та DirectX, які роблять візуалізацію компонентів програми, або виконують іншу низькорівневу обробку. Зокрема, за допомогою низькорівневого інтерфейсу Direct3D, що входить до складу DirectX, відбувається трансляція

Тут також на одному рівні знаходиться бібліотека user32.dll. І хоча вище говорилося, що WPF не використовує цю бібліотеку для рендерингу та візуалізації, проте для низки обчислювальних завдань (що не включають візуалізацію) ця бібліотека продовжує використовуватись.

2.4 Entity Framework Core

Entity Framework представляє ORM-технологію (object-relational mapping – відображення даних на реальні об'єкти) від компанії Microsoft для доступу до даних. Entity Framework Core дозволяє абстрагуватися від самої бази даних та її таблиць та працювати з даними як з об'єктами класом незалежно від типу сховища. Якщо фізично ми оперуємо таблицями, індексами, первинними та зовнішніми ключами, але на концептуальному рівні, який нам пропонує Entity Framework, ми вже працюємо з об'єктами.

Як технологія доступу до даних Entity Framework Core працює поверх платформи .NET і тому може використовуватися на різних платформах стека .NET. Це і стандартні платформи типу Windows Forms, консольні програми, WPF, UWP та ASP.NET Core. При цьому кросплатформова природа EF Core дозволяє задіяти її не тільки на Windows, але і на Linux і Mac OS X.

Оскільки Entity Framework Core працює на основі платформи .NET, він розвивається разом з даною платформою. Поточну версію EF Core - 7.0 було випущено в листопаді 2022 року разом з .NET 7. І технологія продовжує розвиватися.

Entity Framework Core підтримує багато різних систем баз даних. Таким чином, ми можемо через EF Core працювати з будь-яким СУБД, якщо для неї є потрібний провайдер. За замовчуванням на даний момент Microsoft надає ряд вбудованих провайдерів: для роботи з MS SQL Server, SQLite, PostgreSQL. Також є провайдери від сторонніх постачальників, наприклад MySQL.

Також варто відзначити, що EF Core надає універсальний API для роботи з даними. І якщо, наприклад, ми вирішимо змінити цільову СУБД, то основні зміни в проекті стосуватимуться насамперед конфігурації та налаштування підключення до відповідних провайдерів. А код, який безпосередньо працює з даними, отримує дані, додає їх у БД тощо, залишиться тим самим.

Центральною концепцією Entity Framework є поняття сутності чи entity. Сутність визначає набір даних, які пов'язані з певним об'єктом. Тому ця технологія передбачає роботу не з таблицями, а з об'єктами та їх колекціями.

Будь-яка сутність, як і будь-який об'єкт із реального світу, має ряд властивостей. Наприклад, якщо сутність описує людину, ми можемо виділити такі властивості, як ім'я, прізвище, зростання, вік. Властивості необов'язково являють собою прості дані типу int або string, але можуть також представляти і більш комплексні типи даних. І в кожної сутності може бути одна або кілька властивостей, які відрізнятимуть цю сутність від інших і будуть унікально визначати цю сутність. Такі властивості називають ключами.

При цьому сутності можуть бути пов'язані асоціативним зв'язком один-до-багатьом, один-до-одного і багато-багатьом, подібно до того, як у реальній базі даних відбувається зв'язок через зовнішні ключі.

Відмінною рисою Entity Framework Core як технології ORM є використання запитів LINQ для вибірки даних з БД. За допомогою LINQ ми можемо створювати різні запити на вибір об'єктів, у тому числі пов'язаних

різними асоціативними зв'язками. А Entity Framework при виконанні запиту транслює вирази LINQ у вирази, зрозумілі для конкретної СУБД (зазвичай у вирази SQL).

Основна функціональність Entity Framework Core зосереджена у таких пакетах:

1) Microsoft.EntityFrameworkCore:основний пакет EF Core
2) Microsoft.EntityFrameworkCore.SqlServer: представляє функціональність провайдера для Microsoft SQL Server та SQL Azure
3) Microsoft.EntityFrameworkCore.SqlServer.NetTopologySuite: надає підтримку географічних типів (spatial types) для SQL Server
4) Microsoft.EntityFrameworkCore.Sqlite: представляє функціональність провайдера для SQLite і включає нативні бінарні файли для движка бази даних
5) Microsoft.EntityFrameworkCore.Sqlite.Core: представляє функціональність провайдера для SQLite, але на відміну від попереднього пакета не містить нативні бінарні файли для движка бази даних
6) Microsoft.EntityFrameworkCore.Sqlite.NetTopologySuite: надає підтримку географічних типів (spatial types) для SQLite
7) Microsoft.EntityFrameworkCore.Cosmos: представляє функціональність провайдера для Azure Cosmos DB
8) Microsoft.EntityFrameworkCore.InMemory: представляє функціональність провайдера бази даних у пам'яті
9) Microsoft.EntityFrameworkCore.Tools: містить команди EF Core PowerShell для Visual Studio Package Manager Console; застосовується в Visual Studio для міграцій та генерації класів за готовою бд
10) Microsoft.EntityFrameworkCore.Design: містить допоміжні компоненти EF Core, які застосовуються в процесі розробки

11) Microsoft.EntityFrameworkCore.Proxies: зберігає функціональність для так званого "ледачого завантаження" (lazy-loading) та проксі охолодження змін

12) Microsoft.EntityFrameworkCore.Abstractions: містить набір абстракцій EF Core, які не залежать від конкретної СУБД

13) Microsoft.EntityFrameworkCore.Relational: зберігає компоненти EF Core для провайдерів реляційних СУБД

14) Microsoft.EntityFrameworkCore.Analyzers: містить функціонал аналізаторів C# для EF Core

Платформу Entity Framework Core можна застосовувати в різних технологіях стека .NET - консольних програмах, програмах на WinForms, WPF, UWP, веб-додатки ASP.NET і так далі.

РОЗДІЛ 3
СТРУКТУРА ПРОГРАМИ ТА ЇЇ РЕАЛІЗАЦЯ

3.1 Загальна модель програми.

Archer складається з 5 основних модулів:

1) WPF Client – додаток яким буде користуватись клієнт в процесі експлуатації. Додаток створений на технології WPF, що дозволяє доволі просто розробляти дизайн та використовує мінімум потужності ПК. Додаток побудований за допомогою партерна MVVM(Model-View-ViewModel) - шаблон дизайну архітектури застосування. Представлений у 2005 році Джоном Госсманом (John Gossman) як модифікація шаблону Presentation Model. Орієнтований на сучасні платформи розробки, такі як Windows Presentation Foundation:
   - Модель (англ. Model) (так само, як у класичній MVC) є логікою роботи з даними та опис фундаментальних даних, необхідних для роботи програми.
   - Представлення (англ. View) - графічний інтерфейс (вікна, списки, кнопки тощо). Виступає підписником на подію зміни значень властивостей чи команд, що надаються Моделлю Подання. У випадку, якщо в Моделі Уявлення змінилася будь-яка властивість, то вона сповіщає всіх підписників про це і Уявлення, у свою чергу, вимагає оновленого значення властивості в Моделі Уявлення. Якщо користувач впливає на будь-який елемент інтерфейсу, Подання викликає відповідну команду, надану Моделлю Представлення.
   - Модель Подання (англ. ViewModel) - з одного боку, абстракція Представлення, а з іншого - обгортка даних з Моделі, що підлягають зв'язування. Тобто, вона містить Модель, перетворену на Представлення, а також команди, якими може користуватися Представлення, щоб впливати на Модель

2) ExcelReader – бібліотека для роботи з файлами Excel. Цей модуль створений для роботи з специфічними файлами банку, які мають свої специфічні стандарти. Бібліотека створена таким чином, що може бути без жодних проблем замінена на іншу, під потрібний функціонал.
3) DataCore – частина що відповідає за базу даних. Через даний модуль підключена база даних та тут знаходяться класи, що відповідають за представлення даних. За допомогою Entity Framework ми можем дуже просто підключити будь якого провайдера баз даних (MySQL, MSSQL, SQLite та інші…)
4) DataCommand - це ядро програми, що поєднує візуальну частину та базу даних. В цьому модулі знаходяться основні команди для отримання даних та їх обробки та напряму викликаються візуальними клієнтами
5) Console_ExcelReader - це версія клієнта для консолі. Дана версія клієнта потрібна для тестування деякого функціоналу, коли не потрібно чіпати візуальну частину. У випадках коли над проектом працюють тестувальники, ця версія буде дуже зручною для тестування технічних функцій.

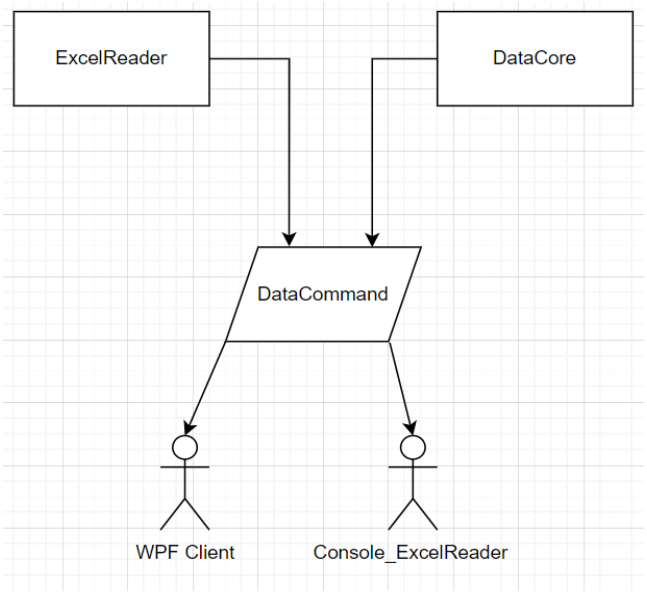

Рис. 3.1. UML схема модулів

3.2 Огляд модуля WPF Client

Модуль WPF Client створений для потреб клієнта, щоб він міг взаємодіяти з базою даних. Структура файлу виглядає наступним чином:

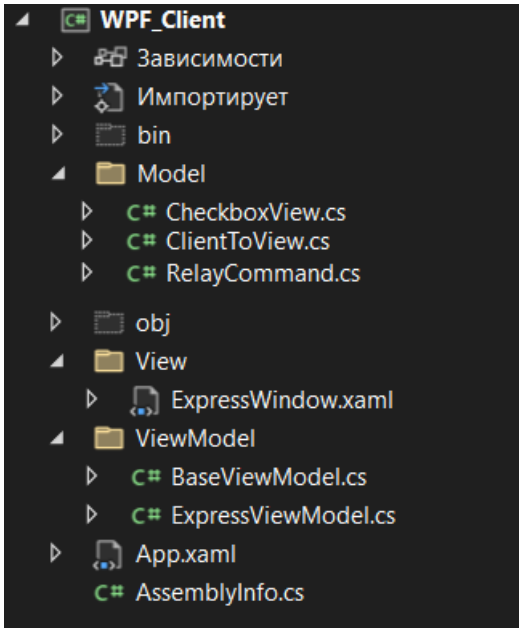

Рис. 3.2. Структура модулю WPF Client

Папка Model містить наступні файли:

1) CheckboxView.cs – спеціальний клас для роботи фільтрів
2) ClientToView.cs - клас для отримання даних з представлення на передачі до моделі
3) RelayCommand.cs – технічний клас для роботи команд в MVVM
4) Папка View містить наступні файли:
5) ExpressWindow.xaml - це наша візуальна частина, яку бачить користувач.
6) Папка Model містить наступні файли:

- BaseViewModel.cs – базовий клас для усіх view model, аби у випадку створення нових сторінок не повторювати код

- ExpressViewModel.cs – клас що містить події для усіх кнопок та контролерів. Він використовує команди з класу DataCommand, тому команди можуть в будь який час без жодних проблем змінені.

3.3 Огляд модуля ExcelReader

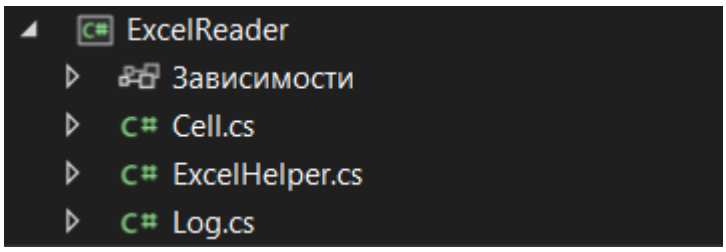

Рис. 3.3. Структура модулю ExcelReader

ExcelReader містить наступні класи:

1) Cell.cs – клас який виконує роль клітини в файлі excel та потрібен для зручного перебору даних.
2) ExcelHelper.cs – основний клас, що виконує перебор даних та пошуку потрібних даних у файлах.
3) Log.cs – клас логу помилок excel, аби розробник або користувач могли перевірити чому завантаження файлів було не успішним.

3.4 Огляд модуля DataCore

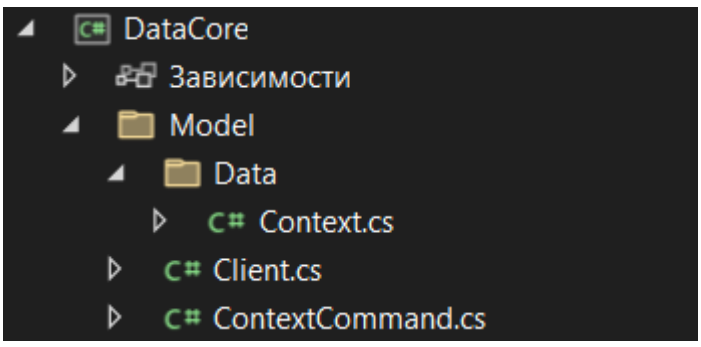

Рис. 3.4. Структура модулю DataCore

ExcelReader містить наступні класи:

1) ContextCommand.cs - містить базові команди до контексту БД, аби зменшити дублювання коду та покращити надійність роботи програми.
2) Client.cs – базова модель клієнта, в якій уся потрібна інформація.
3) Context.cs - містить контекст даних або таблиці в БД

3.5 Огляд модуля CoreCommand

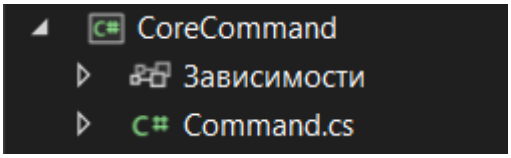

Рис. 3.5. Структура модулю CoreCommand

DataCommand містить наступні класи:

- Command.cs – клас в якому усі потрібні команди для роботи.

Завдяки винесенню усіх важливих команд в окремий клас ми можемо без жодних проблем змінити представлення, або підключити додаток до інших платформ.

3.6 Огляд модуля Console_ExcelReader

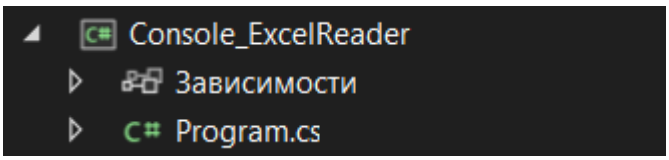

Рис. 3.6. Структура модулю Console_ExcelReader

Console_ExcelReader містить наступні класи:

1) Program.cs – базовий клас для консольного додатку. Потрібен для тестування функціоналу, який не потрібно виводити на WPF додаток.

3.7 Дизайн програми

Для користувача був спеціально створений дизайн:

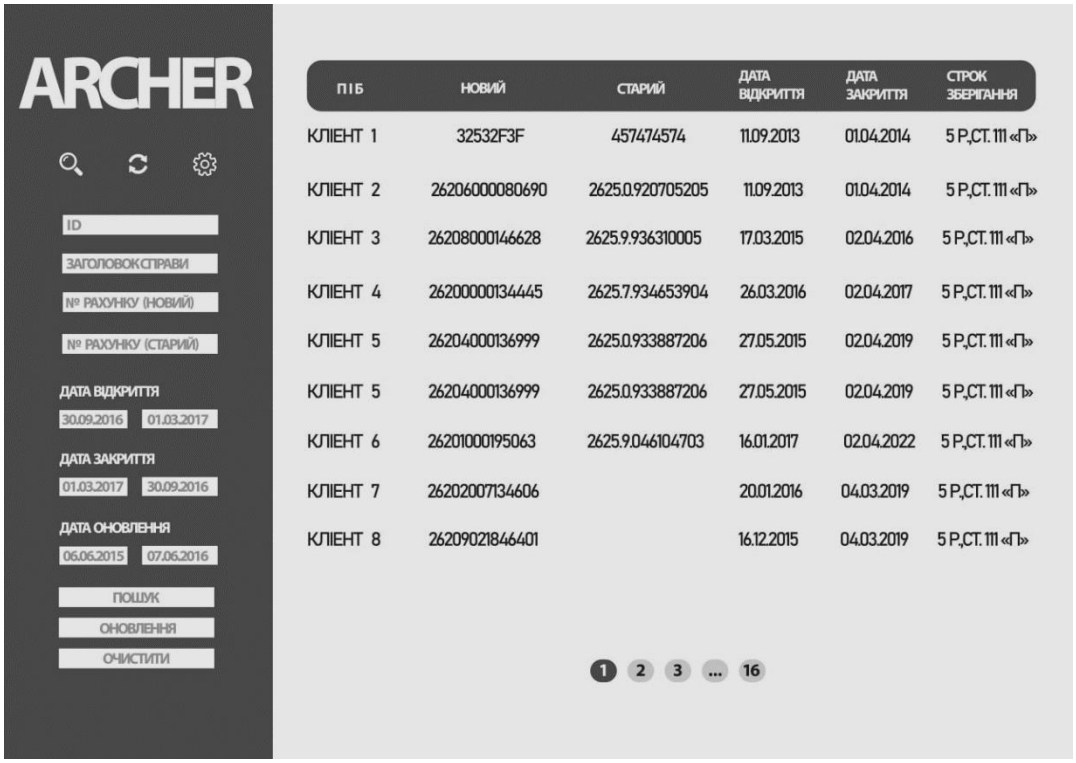

Рис. 3.7. Дизайн додатку

Дизайн був створений за прикладом схожих додатків що працюють с таблицями та великою кількістю даних.

Також існує більш проста версія:

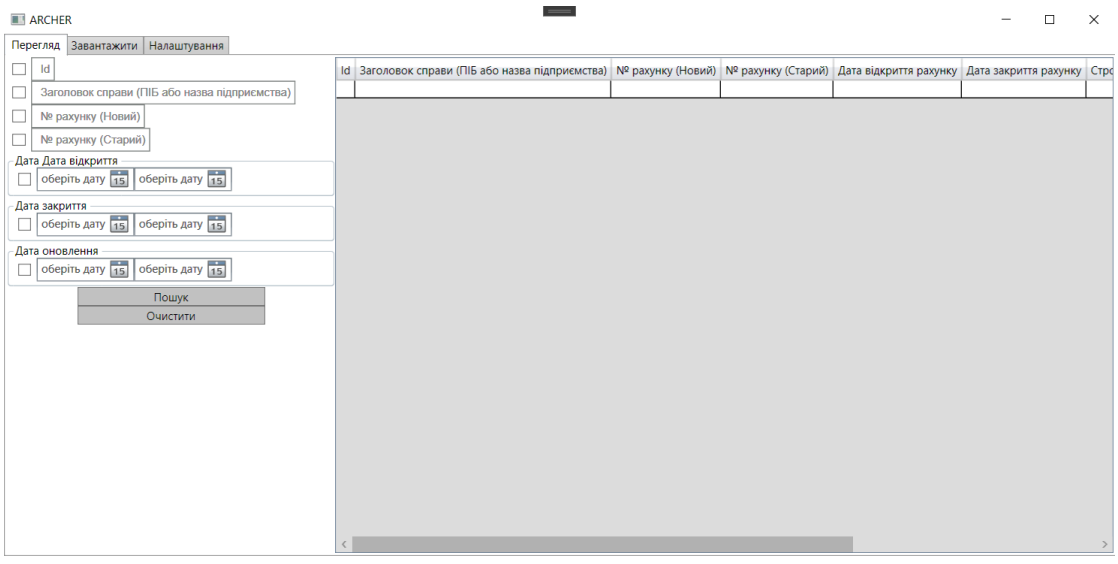

Рис. 3.8. Спрощений дизайн для користувача

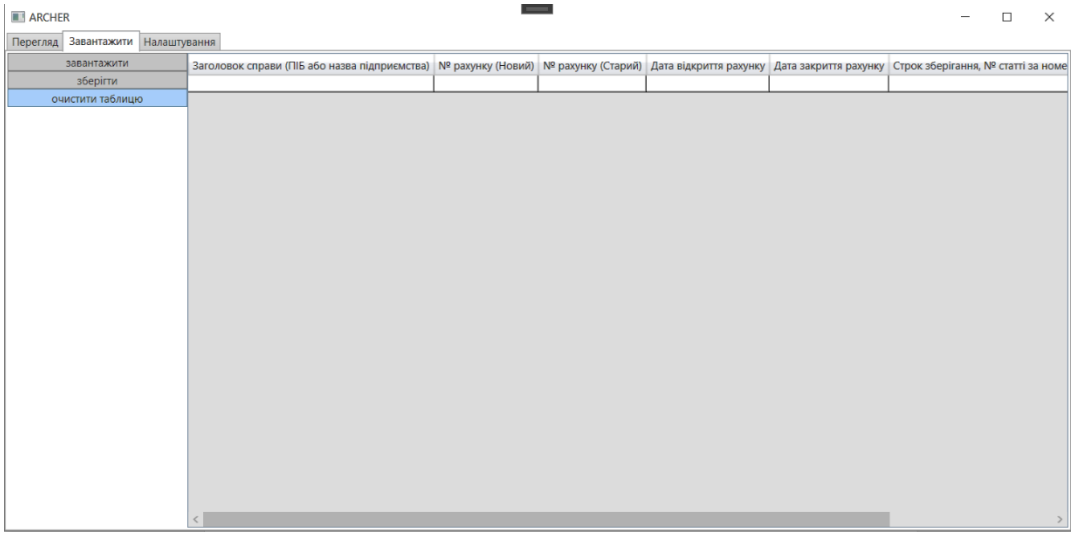

Рис. 3.9. Спрощений дизайн для користувача (2)

3.8 Пошук даних

Для пошуку в програмі реалізовані два методи:

1) Сортування бази даних та пошук з використанням бінарного пошуку
2) Використання Б-дерева в індексі

У першому випадку ідея пошуку полягає в тому, щоб брати елемент посередині між кордонами і порівнювати його з шуканим. Якщо шукане більше (у разі правостороннього — не менше), ніж елемент порівняння, то звужуємо область пошуку так, щоб нова ліва межа дорівнювала індексу середини попередньої області. В іншому випадку привласнюємо це значення правого кордону. Проробляємо цю процедуру доти, доки правий кордон більше лівої більш ніж на 1. Час виконання даного алгоритму $O(2\log n) = O(\log n)$

У другому випадку кількість звернень до диска, необхідне виконання більшості операцій із B-деревом, пропорційно його висоті. Проаналізуємо висоту B-дерева у найгіршому випадку.

Якщо $n \geq 1$, то для B-дерева T с n вузлами та мінімальним ступенем $t \geq 2$ є така нерівність:

$$h \leq log_t \frac{n+1}{2}$$

Корінь B-дерева T містить щонайменше один ключ, а решта вузлів — хоча б t−1 ключів. Так, T, висота якого h, має хоча б 2 вузла на глибині 1, хоча б 2t вузла на глибині 2, хоча б $2t^2$вузла на глибині 3 і так далі, до глибини h, воно має щонайменше $2t^{h-1}$ вузлів . Так, число ключів n задовольняє нерівність:

$$n \geq 1(t-1)\sum_{i-1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1$$

Найпростіше перетворення дає нам нерівність $th \leq (n+1)/2$. Логарифмування на підставі t обох частин нерівності доводить теорему.

Тут бачимо переваги B-дерев над червоно-чорними деревами. Хоча висота дерев росте як $O(\log t)$ в обох випадках (згадаємо, що t - константа), у разі B-дерев основа логарифмів має набагато більше значення. Таким чином, B-дерева вимагають дослідження приблизно в $\log t$ разів меншої кількості вузлів у порівнянні з червоно-чорними деревами в більшості операцій. Оскільки дослідження вузла дерева зазвичай вимагає звернення до диска, кількість дискових операцій при роботі з деревами виявляється суттєво зниженою.

ВИСНОВКИ

1.Розглянуто процес пошуку в предметній області. Визначено основні поняття, що використовуються при цьому.

2.Розглянуті різні способи пошуку та оновлення даних з різними алгоритмами, з урахуванням недоліків.

3.Створено модульну систему, яку можна швидко підстроїти під будь-яку організацію

4.Створено можливість працювати з додатком на усіх платформах при потребі, та з різними типами БД.

Під час виконання випускної кваліфікаційної роботи було отримано такі результати:

- проаналізовано предметну область;
- спроектовано алгоритм роботи і структуру системи;
- розроблена та протестована інформаційна система.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) МЕТОДИЧНІ РЕКОМЕНДАЦІЇ до виконання кваліфікаційних робіт здобувачами другого (магістерського) рівня вищої освіти спеціальностей 121 «Інженерія програмного забезпечення» / / Б.І. Мороз, О.В. Іванченко, О.В. Реута, О.С. Шевцова; М-во освіти і науки України, Нац. техн. ун-т “Дніпровська політехніка”. – Дніпро : НТУ «ДП», 2021. – 56 с.
2) https://habr.com/ru/post/102785/
3) https://lib.iitta.gov.ua/704951/1/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4.%20%D1%80%D0%B5%D0%BA%D0%BE%D0%BC%D0%B5%D0%BD%D0%B4%D0%B0%D1%86%D1%96%D1%97_%D0%A1%D1%83%D1%85%D0%B8%D0%B9%202015.pdf
4) https://practicum.yandex.ru/blog/10-osnovnyh-struktur-dannyh/#:~:text=%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0%20%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85%20%E2%80%94%20%D1%8D%D1%82%D0%BE%20%D1%81%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%20%D0%BE%D1%80%D0%B3%D0%B0%D0%BD%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D0%B8,%D0%9D%D0%B0%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D1%80%2C%20%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2%20%E2%80%94%20%D1%8D%D1%82%D0%BE%20%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0.
5) https://lib.iitta.gov.ua/704951/1/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4.%20%D1%80%D0%B5%D0%BA%D0%BE%D0%BC%D0%B5%D0%BD%D0%B4%D0%B0%D1%86%D1%96%D1%97_%D0%A1%D1%83%D1%85%D0%B8%D0%B9%202015.pdf
6) https://otus.ru/journal/vse-chto-neobhodimo-znat-pro-indeksy-ms-sql/#:~:text=%D0%A7%D1%82%D0%BE%20%D1%82%D0%B0%D0%BA%D0%BE%D0%B5%20%D0%B8%D0%BD%D0%B4%D0%B5%D0%BA%D1%81%D1%8B%20%D0%B2%20sql,%D0%B2%20%D0%BF%D0%BE%

D0%B2%D1%8B%D1%88%D0%B5%D0%BD%D0%B8%D0%B8%20%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%B4%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D0%B8%20sql%20%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D0%BE%D0%B2.

7) http://csharp.net-informations.com/excel/csharp-format-excel.htm
8) https://learn.microsoft.com/en-us/sql/t-sql/statements/create-index-transact-sql?view=sql-server-ver16
9) https://code.4noobz.net/wpf-add-a-watermark-to-a-native-wpf-textbox/
10) https://metanit.com/sharp/tutorial/
11) https://learn.microsoft.com/ru-ru/dotnet/csharp/

ДОДАТОК А

КОД ПРОГРАМИ

```
;
using Microsoft.Win32;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Windows;
using System.Threading.Tasks;
using WPF_Client.Model;
//using CoreCommand;

namespace WPF_Client.ViewModel
{
	internal class ExpressViewModel : BaseViewModel
	{
		public ObservableCollection<Client> ClientsListData { get; set; }
		public ObservableCollection<Client> ClientsListNew { get; set; }

		ClientsData data;

		public ClientToView clientToView { get; set; }
		public CheckboxView checkbox { get; set; }


		public ExpressViewModel()
		{
			clientToView = new ClientToView();

			checkbox = new CheckboxView();
			ClientsListNew = new ObservableCollection<Client>();
			ClientsListData = new ObservableCollection<Client>();
			data = new ClientsData();
		}


		private RelayCommand scanNewExcelTable;
		public RelayCommand ScanNewExcelTable
		{
			get
			{
				return scanNewExcelTable ??
					(scanNewExcelTable = new RelayCommand(async obj =>
					{
						OpenFileDialog file1 = new OpenFileDialog();
						if (file1.ShowDialog() == true)
						{
							List<Client> x = new List<Client>();
							await Task.Run(() => {
									x =
ScanExcelTable(file1.FileName);
							});
							foreach (var nClient in x)
							{
								ClientsListNew.Add(nClient);
							}
							MessageBox.Show($"Сканування
виконано\n3найдено:{ClientsListNew.Count} записів");
						}
					}));
			}
		}
```

```
        private RelayCommand dropNewUserTable;
        public RelayCommand DropNewUserTable
        {
                get
                {
                        return dropNewUserTable ??
                                (dropNewUserTable = new RelayCommand(async obj =>
                                {
                                        ClientsListNew.Clear();
                                        MessageBox.Show("табличка очищена");

                                }));
                }
        }

        private RelayCommand saveNewUsers;
        public RelayCommand SaveNewUsers
        {
                get
                {
                        return saveNewUsers ??
                                (saveNewUsers = new RelayCommand(async obj =>
                                {
                                        await Task.Run(() =>
                                        {
                                                FindClient(ClientsListNew);
                                        });
                                        MessageBox.Show("збережено");
                                        ClientsListNew.Clear();

                                }));
                }
        }


        private RelayCommand findUsersInDatabase;
        public RelayCommand FindUsersInDatabase
        {
                get
                {
                        return findUsersInDatabase ??
                                (findUsersInDatabase = new RelayCommand(async obj =>
                                {
                                        ClientsListData.Clear();
                                        List<Client> x = new List<Client>();
                                        await Task.Run(() =>
                                        {
                                                x = SearchClient(clientToView, checkbox);
                                        });
                                        if (x.Count != 0)
                                        {
                                                foreach (var nClient in x)
                                                {
                                                        ClientsListData.Add(nClient);
                                                }

MessageBox.Show($"знайдено:{ClientsListData.Count} записів");

                                        }
                                        else
                                        {
                                                MessageBox.Show("дані не існують");
                                        }
```

```
                        }));
                }
        }

        private RelayCommand clearSerher;
        public RelayCommand ClearSerher
        {
                get
                {
                        return clearSerher ??
                                (clearSerher = new RelayCommand(async obj =>
                                {
                                        clientToView.ClearMe();
                                        NotifyPropertyChanged("clientToView");

                                }));
                }
        }


        private RelayCommand dropDb;
        public RelayCommand DropDB
        {
                get
                {
                        return dropDb ??
                                (dropDb = new RelayCommand(async obj =>
                                {
                                        MessageBox.Show("починаю видаляти бд");
                                        await Task.Run(() =>
                                        {
                                                data.DropDataBase();
                                        });
                                        MessageBox.Show("база даних видалена");

                                }));
                }
        }



        List<Client> ScanExcelTable(string filePath)
        {
                using (ExcelHelper helper = new ExcelHelper(filePath))
                {
                        List<Client> newClientList = new List<Client>();
                        for (int i = 1; i <= helper.GetWorksheetsCount(); i++)
                        {

                                Cell? start_cell = helper.FindStartTable(i);
                                if (start_cell == null)
                                {
                                        break;
                                }

                                for (int row = start_cell.Y; row < 300; row++)
                                {
                                        if
(string.IsNullOrEmpty(Convert.ToString(helper.ReadCell(row, 1, i))))
                                        {
                                                break;
                                        }
                                        newClientList.Add(new Client(
                                                helper.ReadCell(row, 2, i),
```

```
                                        helper.ReadCell(row, 3, i),
                                        helper.ReadCell(row, 4, i),
                                        helper.ReadCell(row, 5, i),
                                        helper.ReadCell(row, 6, i),
                                        helper.ReadCell(row, 7, i),
                                        helper.ReadCell(row, 8, i)));
                                };

                        }
                        return newClientList;
                }


        }

        List<Client> SearchClient(ClientToView client, CheckboxView checkbox)
        {
                bool flag = false;
                var r = from p in data.GetClients()
                        select (p);

                if (client.Id != null & checkbox.chId != true)
                {
                        r = r.Where(b => b.Id == client.Id); flag = true;
                }
                if (!string.IsNullOrEmpty(client.Name) & checkbox.chName != true)
                {
                        r = r.Where(b => b.Name == client.Name); flag = true;
                }
                if (!string.IsNullOrEmpty(client.NewAccountNumber) &
checkbox.chNewAccN != true)
                {
                        r = r.Where(b => b.NewAccountNumber ==
client.NewAccountNumber); flag = true;
                }
                if (!string.IsNullOrEmpty(client.OldAccountNumber) &
checkbox.chOldAccN != true)
                {
                        r = r.Where(b => b.OldAccountNumber ==
client.OldAccountNumber); flag = true;
                }

                if (checkbox.chDateOp != true)
                {
                        if (client.DateOpenMin != null)
                        {
                                r = r.Where(b => b.DateOpen > client.DateOpenMin);
flag = true;
                        }
                        if (client.DateOpenMax != null)
                        {
                                r = r.Where(b => b.DateOpen < client.DateOpenMax);
flag = true;
                        }
                }
                if (checkbox.chDateCl != true)
                {
                        if (client.DateCloseMin != null)
                        {
                                r = r.Where(b => b.DateClose > client.DateCloseMin);
flag = true;
                        }
                        if (client.DateCloseMax != null)
                        {
```

```
                                    r = r.Where(b => b.DateClose < client.DateCloseMax);
flag = true;
                              }
                        }
                        if (checkbox.chDateUp != true)
                        {
                              if (client.DateUpdateMin != null)
                              {
                                    r = r.Where(b => b.DateUpdate > client.DateUpdateMin);
flag = true;
                              }
                              if (client.DateUpdateMax != null)
                              {
                                    r = r.Where(b => b.DateUpdate < client.DateUpdateMax);
flag = true;
                              }
                        }


                        if (flag == true)
                        {
                              return r.ToList();
                        }
                        else
                        {
                              return new List<Client>();
                        }

                        return new List<Client>();
                  }

                  void FindClient(ObservableCollection<Model.Client> clientsList)
                  {
                        bool flag = false;
                        List<Client> clientsQnic = new List<Client>();

                        List<Model.Client> clientsData = data.GetClients();

                        for (int i = 0; i < clientsList.Count; i++)
                        {
                              for (int y = 0; y < clientsData.Count; y++)
                              {
                                    if (clientsList[i].OldAccountNumber ==
clientsData[y].OldAccountNumber & clientsList[i].NewAccountNumber ==
clientsData[y].NewAccountNumber)
                                    {
                                          data.UpdateClient(clientsData[y],
clientsList[i]);
                                          flag = true;
                                          break;
                                    }

                                    if (clientsList[i].OldAccountNumber ==
clientsData[y].NewAccountNumber)
                                    {
                                          data.UpdateClient(clientsData[y],
clientsList[i]);
                                          flag = true;
                                          break;
                                    }
                              }
                              if (flag == true)
                              {
                                    flag = false;
                                    continue;
```

```
                        }
                        clientsQnic.Add(clientsList[i]);
                }
                data.AddClient(clientsQnic);

        }

    }
}

using Microsoft.Office.Interop.Excel;
using System.Runtime.InteropServices;
using Excel = Microsoft.Office.Interop.Excel;

namespace ExcelReader
{
    public class ExcelHelper : IDisposable
    {
        Log log;
        private Application _excel;
        private Workbook? _workbook;
        private string? _filePath;

        public ExcelHelper()
        {
            log = new();
            _excel = new Application();

        }
        public ExcelHelper(string path)
        {
            log = new();
            _excel = new Application();

            Open(path);

        }

        public void Dispose()
        {
            try
            {
                _workbook?.Close(0);
                _excel.Quit();

                Marshal.ReleaseComObject(_workbook);
                Marshal.ReleaseComObject(_excel);

            //    Console.Beep();
            }
            catch (Exception ex)
            {
                log.Set(ex.Message);
            }

        }

        public void Close()
        {

        }

        public bool Open(string path)
        {
            try
```

```
                {
                        if (File.Exists(path))
                        {
                                _workbook = _excel.Workbooks.Open(path);
                        }
                        else
                        {
                                _workbook = _excel.Workbooks.Add();
                                _filePath = path;

                        }
                        return true;
                }
                catch (Exception ex)
                {
                        log.Set(ex.Message);
                        return false;
                }
        }

        public bool Set(string colum, int row, string data)
        {
                try
                {
                        ((Excel.Worksheet)_excel.ActiveSheet).Cells[row, colum] =
data;

                        return true;
                }
                catch (Exception ex)
                {
                        log.Set(ex.Message);
                        return false;
                }


        }

        public void Save()
        {
                try
                {
                        if (_filePath == null)
                                _workbook?.Save();
                        else
                        {
                                _workbook?.SaveAs(_filePath);
                                _filePath = null;
                        }
                }
                catch (Exception ex)
                {
                        log.Set(ex.Message);
                }
        }

        public int? GetWorksheetsCount()
        {
                return _workbook?.Worksheets.Count;
        }

        public dynamic ReadCell(int row, int colum, int worksheets_id)
        {
                try
                {
```

```
                        Worksheet worksheet = _workbook.Worksheets[worksheets_id];
                        return worksheet.Cells[row, colum].Value;
                    }
                    catch (Exception ex)
                    {
                        log.Set(ex.Message);
                        return ex.Message;
                    }
                }

                public Cell? FindStartTable(int docList, string simbol="1")
                {
                    for (int row = 1; row <= 30; row++)
                    {
                        if (Convert.ToString(ReadCell(row, 1, docList)) == simbol)
                        {
                            return new Cell { Y = row, X = 1, IdList = docList };
                        }
                    }
                    return null;
                }

        }
}



using System;

namespace WPF_Client.Model
{
        public class Client
        {
                public int Id { get; set; }
                public string Name { get; set; }
                public string NewAccountNumber { get; set; }
                public string? OldAccountNumber { get; set; }
                public DateTime? DateOpen { get; set; }
                public DateTime? DateClose { get; set; }
                public string? DocInfo { get; set; }
                public string? Comment { get; set; }
                public DateTime? DateUpdate { get; set; }

                public Client() { }
                public Client(dynamic name, dynamic new_account_number, dynamic
old_account_number, dynamic date_open, dynamic date_close, dynamic doc_info, dynamic
comment)
                {
                    Name = Convert.ToString(name);
                    NewAccountNumber = Convert.ToString(new_account_number);
                    OldAccountNumber = Convert.ToString(old_account_number);
                    DateOpen = Convert.ToDateTime(date_open);
                    DateClose = Convert.ToDateTime(date_close);
                    DocInfo = Convert.ToString(doc_info);
                    Comment = Convert.ToString(comment);
                }

        }
}
```

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Диплом_Кісенко.doc	Пояснювальна записка роботи. Документ Word.
Диплом_Кісенко.pdf	Пояснювальна записка роботи в форматі PDF
Програма	
Program.rar	Архів. Містить коди програми і откомпільовану програму
Презентація	
Презентація_Кісенко.ppt	Презентація роботи