

Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття ступеня вищої освіти бакалавра
зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Розробка інструментальних засобів для індексації жорсткого диску
та швидкого пошуку файлів

Засвідчую, що в цій
кваліфікаційній роботі немає
запозичень із праць інших
авторів без відповідних
посилань.

Студент гр. ІПЗ–22–1

_____ / М. О. Ющенко /

Керівник кваліфікаційної
роботи

_____ / Н. О. Карабут /

В.о. завідувача кафедри

_____ / А. М. Стрюк /

Кривий Ріг

2026

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: бакалавр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

В.о. зав. кафедри

_____ А. М. Стрюк

«_____» _____ 2026 р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ–22–1 Ющенко Матвію Олександровичу

1. Тема: «Розробка інструментальних засобів для індексації жорсткого диску та швидкого пошуку файлів» затверджена наказом по КНУ № 284с від «28» травня 2026 р.
2. Термін подання студентом закінченої роботи: «31» травня 2026 р.
3. Вихідні дані по роботі: розроблений програмний комплекс має забезпечити швидкий та зручний пошук файлів в операційній системі Windows.
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): провести аналіз існуючих на ринку аналогічних програмних продуктів, обґрунтувати функціонал розроблюваної системи, створити програмне забезпечення, здійснити тестування розробленого додатку.
5. Перелік ілюстративного матеріалу: функціональна схема, блок–схема алгоритму, зображення екранних форм додатку.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Розгляд літературних джерел та пошук інтернет-ресурсів з заданої тематики	04.01.26 – 24.01.26
2	Аналіз існуючих методів вирішення проблеми	25.01.26 – 07.02.26
3	Формулювання актуальності роботи і постановка завдань	08.02.26 – 18.02.26
4	Оформлення матеріалів першого розділу роботи	18.02.26 – 02.03.26
5	Створення функціональної схеми додатку	03.03.26 – 15.03.26
6	Оформлення матеріалів другого розділу роботи	16.04.26 – 09.04.26
7	Розробка алгоритму роботи, інтерфейсу програмного забезпечення, програмних модулів	10.04.26 – 01.05.26
8	Оформлення матеріалів третього розділу роботи та додатків	02.05.26 – 07.05.26
9	Тестування розробленої програми	08.05.26 – 15.05.26
10	Оформлення пояснювальної записки	16.05.26 – 30.05.26

Дата видачі завдання: «03» січня 2026 р.

Студент: _____ / М. О. Ющенко /

Керівник роботи: _____ / Н. О. Карабут /

РЕФЕРАТ

ШВИДКИЙ ПОШУК ФАЙЛІВ, IN-MEMORY ІНДЕКС, БАГАТОПОТОКОВІСТЬ, ПОТОКОБЕЗПЕЧНІ КОЛЕКЦІЇ, ФІЛЬТРАЦІЯ ДАНИХ, ФАЙЛОВА СИСТЕМА, АЛГОРИТМИ ПОШУКУ.

Пояснювальна записка: 67 с., 11 рис., 1 дод., 15 джерел.

Об'єкт дослідження: процес пошуку, індексації та обробки метаданих файлової системи в операційній системі Windows.

Предмет дослідження: методи, алгоритми та програмні засоби розробки високопродуктивних додатків для пошуку інформації.

Мета роботи: створення ефективного, стабільного та зручного програмного забезпечення для швидкого пошуку файлів за складними критеріями з мінімальним навантаженням на апаратні ресурси комп'ютера.

Методи дослідження: об'єктно-орієнтоване проєктування, методи асинхронного та багатопотокового програмування, алгоритми обходу деревоподібних структур, методи проєктування графічних користувацьких інтерфейсів.

У кваліфікаційній роботі проаналізовано недоліки існуючих інструментів пошуку та обґрунтовано необхідність розробки спеціалізованого рішення. Спроектовано багаторівневу архітектуру програмного продукту на базі платформи .NET та технології Windows Forms.

Розроблено асинхронний сканер файлової системи, який функціонує у фоновому потоці та оснащений механізмом безпечного перехоплення винятків, що запобігає аварійному завершенню роботи.

Створено пошуковий рушій, здатний здійснювати каскадну фільтрацію за типом файлу, розширенням, часовим діапазоном та розміром. Розроблено ергономічний користувацький інтерфейс із застосуванням механізмів Drag-and-Drop, відкладеного пошуку та розумної валідації параметрів.

ABSTRACT

FAST FILE SEARCH, IN-MEMORY INDEX, MULTITHREADING, THREAD-SAFE COLLECTIONS, DATA FILTERING, FILE SYSTEM, SEARCH ALGORITHMS.

Explanatory note: 67 pages, 11 figures, 1 appendix, 15 sources.

Object of study: the process of searching, indexing, and processing file system metadata in the Windows operating system.

Subject of study: methods, algorithms, and software tools for developing high-performance applications for information search.

Purpose of the work: to create an efficient, stable, and user-friendly software application for fast file searching based on complex criteria with minimal load on computer hardware resources.

Research methods: object-oriented design, asynchronous and multithreaded programming methods, tree structure traversal algorithms, and graphical user interface design methods.

The qualification thesis analyzes the shortcomings of existing search tools and justifies the need to develop a specialized solution. A multi-layered architecture of the software product was designed based on the .NET platform and Windows Forms technology.

An asynchronous file system scanner has been developed, which operates in a background thread and is equipped with a safe exception handling mechanism that prevents abnormal termination (crashes) of the application.

A flexible search engine was created, capable of performing cascade filtering by file type, extension, time range, and size. An ergonomic user interface was developed using Drag-and-Drop mechanisms, delayed search, and smart parameter validation.

ЗМІСТ

ВСТУП.....	8
1 ВИЗНАЧЕННЯ СТАНУ СПРАВ У ПИТАННІ ШВИДКОГО ПОШУКУ ФАЙЛІВ.....	10
1.1 Розгляд досліджуваної проблеми.....	10
1.2 Аналіз існуючих рішень для індексації жорсткого диску та швидкого пошуку файлів.....	12
1.2.1 Windows Search	12
1.2.2 Everything	13
1.2.3 Agent Ransack	15
1.3 Результат огляду програмного забезпечення для швидкого пошуку файлів.....	17
2 ПРОЕКТУВАННЯ ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ДЛЯ ІНДЕКСАЦІЇ ЖОРСТКОГО ДИСКУ ТА ШВИДКОГО ПОШУКУ ФАЙЛІВ	19
2.1 Постановка цілей та формування вимог до програмних засобів для індексації жорсткого диску та швидкого пошуку файлів.....	19
2.2 Архітектура програмних засобів для індексації жорсткого диску та швидкого пошуку файлів.....	20
2.3 Функціональна схема програмних засобів для індексації жорсткого диску та швидкого пошуку файлів	22
2.4 Проєктування підсистеми сканування файлової системи.....	25
2.5 Організація in-memory індексу для збереження метаданих.....	26
2.6 Розробка пошукового рушія та механізмів фільтрації.....	27
2.7 Проєктування користувацького інтерфейсу та взаємодії (UX/UI)	29
3 РОЗРОБКА ПРОГРАМНИХ ЗАСОБІВ ДЛЯ ІНДЕКСАЦІЇ ЖОРСТКОГО ДИСКУ ТА ШВИДКОГО ПОШУКУ ФАЙЛІВ	31

3.1 Алгоритм роботи програмних засобів для індексації жорсткого диску та швидкого пошуку файлів.....	31
3.2 Реалізація підсистеми асинхронного сканування файлової системи	33
3.3 Керівництво користувача та опис роботи програми	35
ВИСНОВКИ	40
ПЕРЕЛІК ПОСИЛАНЬ	42
Додаток А - Програмний код.....	43

ВСТУП

Сьогодні обсяги цифрових даних, які щодня обробляє звичайний користувач або розробник, зростають стрімкими темпами. Жорсткі диски та твердотільні накопичувачі об'ємом у кілька терабайтів стали звичним стандартом навіть для базових комп'ютерних збірок. На цих носіях зберігаються сотні тисяч, а іноді й мільйони різних файлів: вихідні коди програм, підключені бібліотеки, мультимедійні матеріали, робочі документи, резервні копії та системні логи.

За таких умов класична ієрархічна модель організації інформації, коли файли ретельно сортуються і розкладаються по вкладених папках, стає неефективною. Людина просто фізично не здатна запам'ятати точне розташування кожного документа через місяці чи роки після його створення. Тому пошуковий запит стає основним способом взаємодії з файловою системою.

Швидкість знаходження потрібної інформації безпосередньо впливає на продуктивність. Замість того, щоб фокусуватися на написанні коду чи аналізі даних, спеціаліст змушений чекати, поки система просканує сотні каталогів. Це створює когнітивне навантаження та порушує робочий процес. Отже, потреба у швидких та надійних інструментах для навігації по локальному диску є доволі гострою.

Штатні засоби пошуку, вбудовані у сучасні операційні системи, часто не справляються з такими навантаженнями. Зазвичай вони покладаються на фонові служби індексації, які постійно моніторять диск і створюють приховані бази даних. Цей процес непомітно споживає ресурси процесора та оперативної пам'яті, а також значно навантажує підсистему введення-виведення. Якщо ж виникає потреба знайти файл у зовнішній директорії або на флеш-накопичувачі, який не потрапив до бази індексу, система починає повільний послідовний обхід дерева каталогів. На великих дисках це призводить до тривалих затримок і навіть до виникнення «фризів» у графічному інтерфейсі.

Існуючі сторонні рішення також мають певні недоліки, оскільки часто ґрунтуються на технічних компромісах. Частина з них досягає високої швидкості завдяки прямому читанню службових таблиць файлової системи на найнижчому рівні. Однак такий підхід прив'язує програму до конкретного формату розмітки диска, вимагає постійних прав адміністратора та ускладнює роботу з мережевими дисками. Інша категорія утиліт пропонує глибокий пошук, але має надто складний інтерфейс. У таких програмах для застосування елементарних фільтрів (наприклад, щоб знайти відеофайл певного розміру, створений минулого тижня) користувачеві доводиться вводити специфічні текстові команди замість використання звичних візуальних перемикачів [1].

З огляду на це, розробка власного спеціалізованого додатка для пошуку файлів є актуальним та важливим завданням.

1 ВИЗНАЧЕННЯ СТАНУ СПРАВ У ПИТАННІ ШВИДКОГО ПОШУКУ ФАЙЛІВ

1.1 Розгляд досліджуваної проблеми

Розробка будь-якого програмного забезпечення починається з детального вивчення предметної області та інструментів, які вже закріпилися на ринку. Процес створення ефективної системи пошуку файлів вимагає розуміння того, як еволюціонували підходи до зберігання інформації та які архітектурні обмеження накладають сучасні операційні системи.

Історично файлові системи мали досить просту структуру, а кількість об'єктів на жорстких дисках вимірювалася тисячами. У таких умовах звичайний рекурсивний обхід дерева каталогів працював швидко. Програма просто відкривала папку, читала її вміст і переходила до наступної. Проте з поширенням сучасних стандартів (наприклад, NTFS) та збільшенням ємності носіїв ситуація кардинально змінилася. Сучасна операційна система оперує мільйонами файлів, кожен з яких супроводжується розширеним набором метаданих: точними датами створення та модифікації, правами доступу власників, атрибутами безпеки тощо.

Відповідно, класичний синхронний підхід до пошуку вичерпав себе. Якщо програма намагається прочитати великий масив директорій у головному потоці виконання, це неминуче призводить до зависання графічного інтерфейсу. Вікно додатку перестає реагувати на дії користувача і операційна система позначає його статусом «Не відповідає». Це змусило розробників шукати нові програмні парадигми, переходячи до асинхронного читання та винесення важких дискових операцій у фонові потоки.

Одним із компромісних рішень стало попереднє індексування. Суть цього методу полягає у тому, що спеціальна служба заздалегідь, у фоновому режимі, збирає інформацію про всі файли та формує оптимізовану локальну базу даних. Коли користувач вводить пошуковий запит, пошук відбувається не на фізичному жорсткому диску, а саме в цій базі даних. Хоча це

кардинально скорочує час відгуку, такий підхід породжує нові виклики. По-перше, сама база даних починає займати гігабайти дискового простору. По-друге, процес постійного моніторингу змін (створення, видалення чи переміщення файлів) створює фонове навантаження на підсистему введення-виведення та оперативну пам'ять комп'ютера [2].

Альтернативний технічний шлях полягає у низькорівневому читанні службових структур файлової системи. Замість того, щоб викликати стандартні функції Windows API для обходу кожної директорії, програма звертається напямую до Master File Table (MFT) – головної таблиці, де зберігаються записи про всі існуючі файли на диску. Це дозволяє отримати повний зріз файлової системи за лічені секунди. Проте реалізація такого алгоритму має обмеження: вона жорстко прив'язує код до однієї конкретної розмітки, ігнорує мережеві диски та вимагає від користувача запуску програми з правами адміністратора, що не завжди можливо у корпоративному середовищі.

Окрім суто технічних перепон, розробники стикаються з проблемою проєктування користувацького досвіду (UX). Уявімо ситуацію: програма за секунду знайшла двадцять тисяч документів, що містять слово «звіт». Для користувача такий результат є марним, адже знайти потрібний файл серед тисяч інших вручну неможливо. Тому сучасний інструмент повинен не лише швидко читати диск, а й надавати зручні візуальні засоби для фільтрації результатів (за точним розміром, періодом створення, розширенням) без необхідності вивчення специфічного синтаксису консольних команд [3].

Усвідомлення цих архітектурних компромісів є важливим для проєктування додатку. Щоб уникнути типових помилок, детально проаналізуємо три найпопулярніші класи пошукових інструментів, кожен з яких використовує свій підхід: вбудовані засоби Windows, утиліти на базі читання MFT та спеціалізовані пошукові рушії.

1.2 Аналіз існуючих рішень для індексації жорсткого диску та швидкого пошуку файлів

1.2.1 Windows Search

Це базовий засіб пошуку, який вбудований в операційну систему від Microsoft. Найчастіше користувачі взаємодіють з ним через рядок пошуку у меню «Пуск» або прямо у вікні Провідника (File Explorer). Архітектура цього інструменту спирається на системну службу Windows Search Service. Її головне завдання – працювати у фоні, сканувати диск, зчитувати атрибути файлів (а часто і їхній вміст) та записувати ці дані у спеціальну приховану базу даних, яка і називається індексом [4].

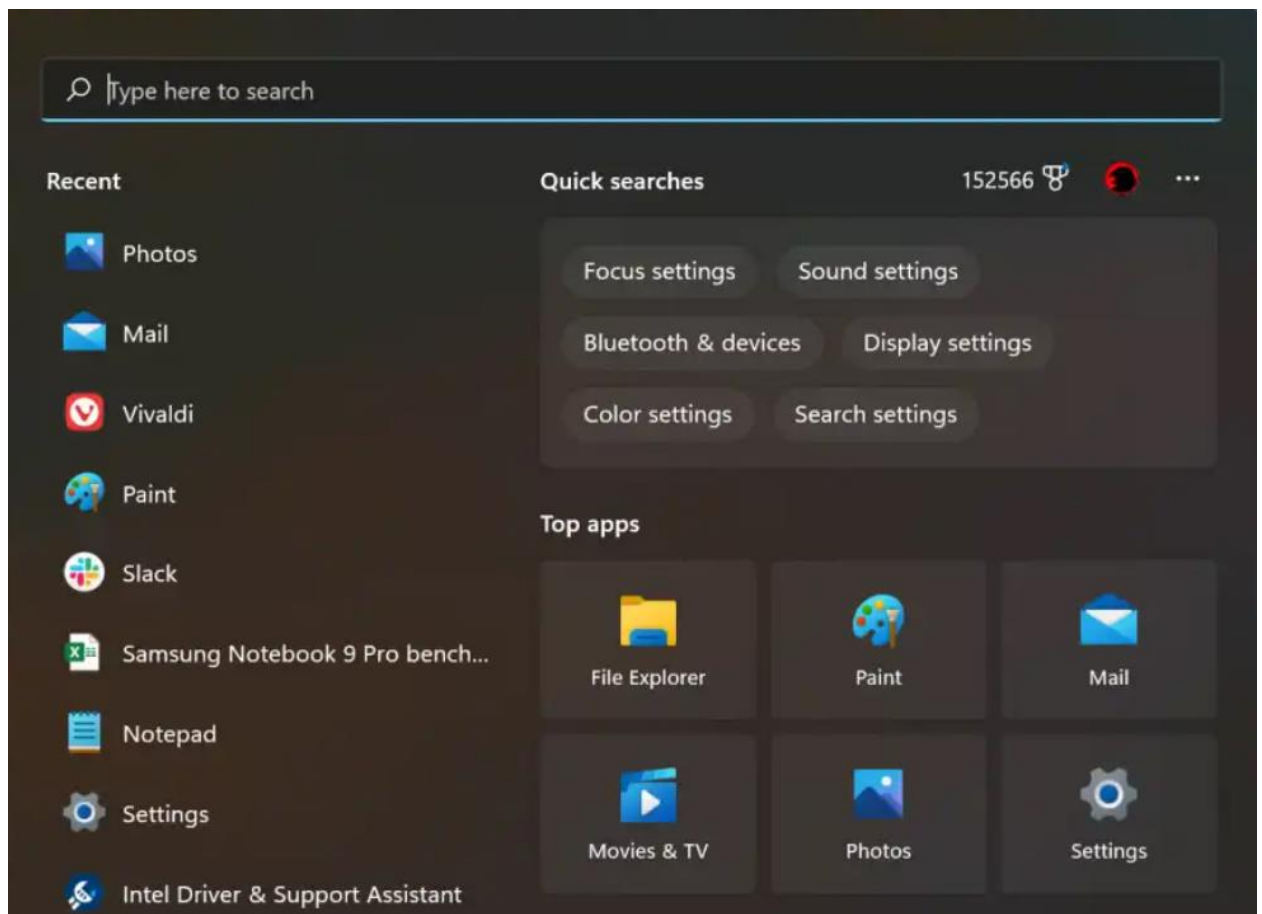


Рисунок 1.1 – Windows Search

Теоретично наявність такої бази має гарантувати миттєвий результат. Але на практиці користувачі регулярно стикаються з критичними падіннями

продуктивності. Причина криється у налаштуваннях операційної системи: щоб не перевантажувати комп'ютер, Windows за замовчуванням індексує лише дуже вузький перелік директорій. Зазвичай це папка поточного користувача, Робочий стіл та завантаження.

Якщо потрібний файл знаходиться на іншому логічному диску (наприклад, D:), на підключеній флешці або у специфічній робочій папці, якої немає в списку індексації, поведінка Windows Search кардинально змінюється. Система відмовляється від використання бази і переходить у режим прямого послідовного читання диска. Програма починає рекурсивно відкривати кожен вкладену папку і перевіряти кожен файл.

Для розробника це стає суттєвою проблемою. Наприклад, стандартний проєкт з використанням веб-технологій може містити папку `node_modules`, де лежать десятки тисяч дрібних файлів. Спроба виконати пошук у такій директорії за допомогою стандартного Провідника призводить до того, що зелена смуга прогресу повільно повзе хвилинами, навантаження на диск зростає до 100%, а саме вікно Провідника часто блокується і перестає реагувати на кліки [5].

Ще одна відчутна проблема – це нестабільність самої фонові служби. Процес оновлення індексу (відомий у системі як `SearchIndexer.exe`) може запуснитися в незручний момент, забираючи на себе значну частину ресурсів процесора та підсистеми пам'яті.

Отже, Windows Search добре справляється з пошуком базових документів у стандартних папках. Проте через надмірне споживання ресурсів фонові службою та вкрай повільний механізм обходу неіндексованих локацій, цей інструмент не підходить для швидкої та інтенсивної роботи з великими обсягами файлів.

1.2.2 Everything

Якщо системний пошук робить ставку на повільну фонову індексацію, то сторонні розробники часто шукають обхідні шляхи на нижчому рівні

операційної системи. Яскравим прикладом такого підходу є утиліта Everything [6] від компанії Voidtools.

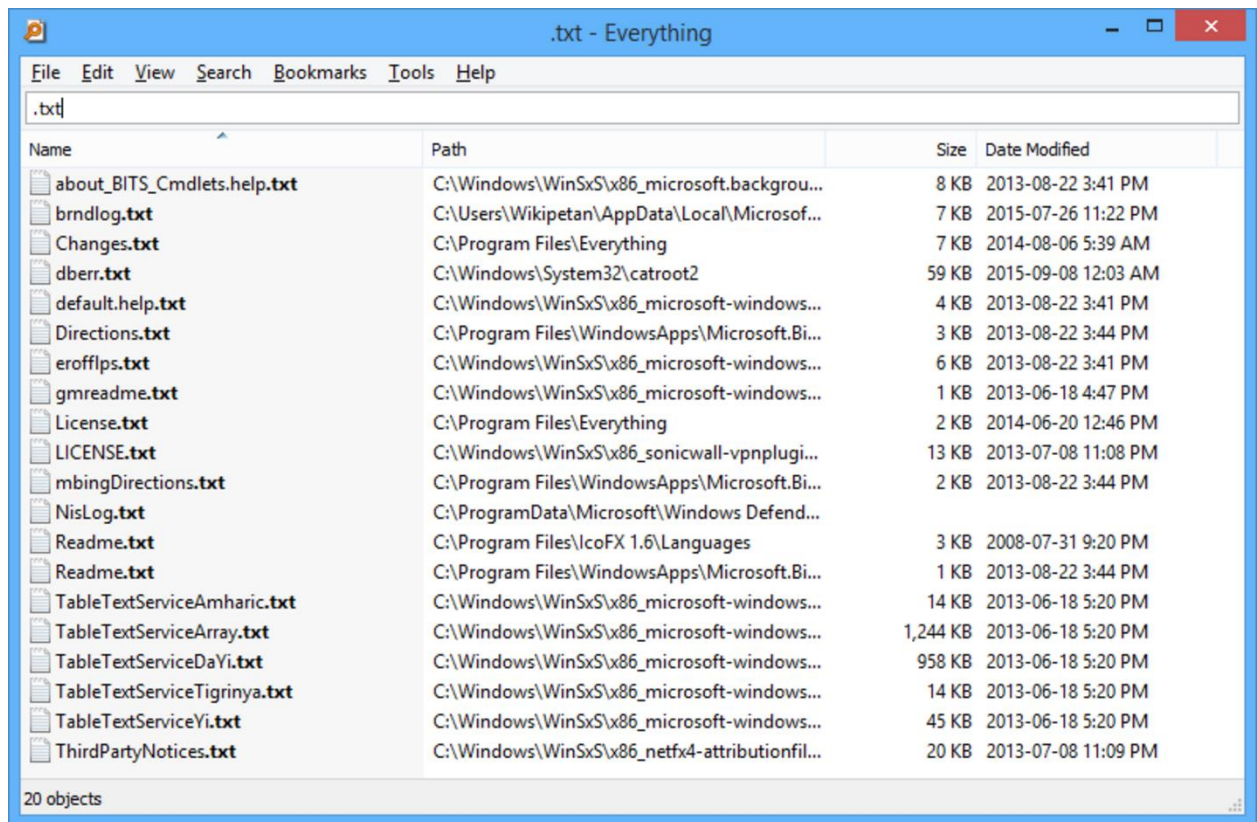


Рисунок 1.2 – Everything

Цей програмний продукт є найбільш відомою альтернативою вбудованим засобам Windows, а його головна технічна перевага – практично миттєва видача результатів пошуку відразу після запуску. Така швидкодія досягається завдяки відмові від стандартних функцій Windows API для рекурсивного обходу директорій. Замість цього програма звертається безпосередньо до MFT – головної файлової таблиці, яка є ядром файлової системи NTFS. MFT містить метадані про кожен файл та каталог на томі, тому пряме зчитування цієї таблиці дозволяє побудувати повний зріз диска за лічені секунди [7].

Проте архітектура, побудована навколо MFT, має кілька суттєвих обмежень. По-перше, цей метод працює виключно з локальними дисками,

розміченими у форматі NTFS (або ReFS). Якщо користувачеві потрібно знайти дані на зовнішньому накопичувачі з файловою системою FAT32/exFAT або на мережевому диску, Everything змушений перемикатися на класичний, повільний алгоритм обходу папок. По-друге, для прямого доступу до службових областей диска програма вимагає обов'язкового запуску з правами адміністратора. Це створює незручності та потенційні ризики безпеки, особливо в умовах обмежених корпоративних мереж.

Друга, і мабуть, найбільш відчутна для кінцевого користувача проблема Everything – це відсутність інтуїтивно зрозумілої системи розширених фільтрів. Розробники програми пішли шляхом створення власної мови пошукових запитів. Інтерфейс утиліти складається переважно з одного текстового рядка.

Для того щоб виконати специфічний пошук – наприклад, знайти всі відеофайли розміром понад 1 ГБ, які були створені минулого місяця, – користувачеві доведеться скласти і ввести текстовий запит на кшталт `ext:mp4;mkv;avi size:>1gb dm:lastmonth`. З точки зору проєктування користувацького досвіду (UX), такий підхід порушує базовий принцип наочності [8]. Замість того, щоб обирати потрібні параметри мишкою за допомогою випадających списків чи повзунків, людина змушена тримати в голові синтаксис спеціальних команд або постійно звертатися до довідки.

Підсумовуючи, Everything блискуче вирішує проблему швидкості на локальних NTFS-дисках, але через високий поріг входження та специфічний «консольний» підхід до фільтрації результатів, ця програма залишається інструментом переважно для технічних спеціалістів, а не масового користувача.

1.2.3 Agent Ransack

Третім показовим прикладом на ринку пошукових утиліт є Agent Ransack [9] (також відомий як FileLocator Lite) від розробників з Mythicsoft. Якщо попередні інструменти робили ставку на швидке поверхневе сканування

файлової системи, то цей програмний продукт був створений переважно для глибокого аналізу даних.

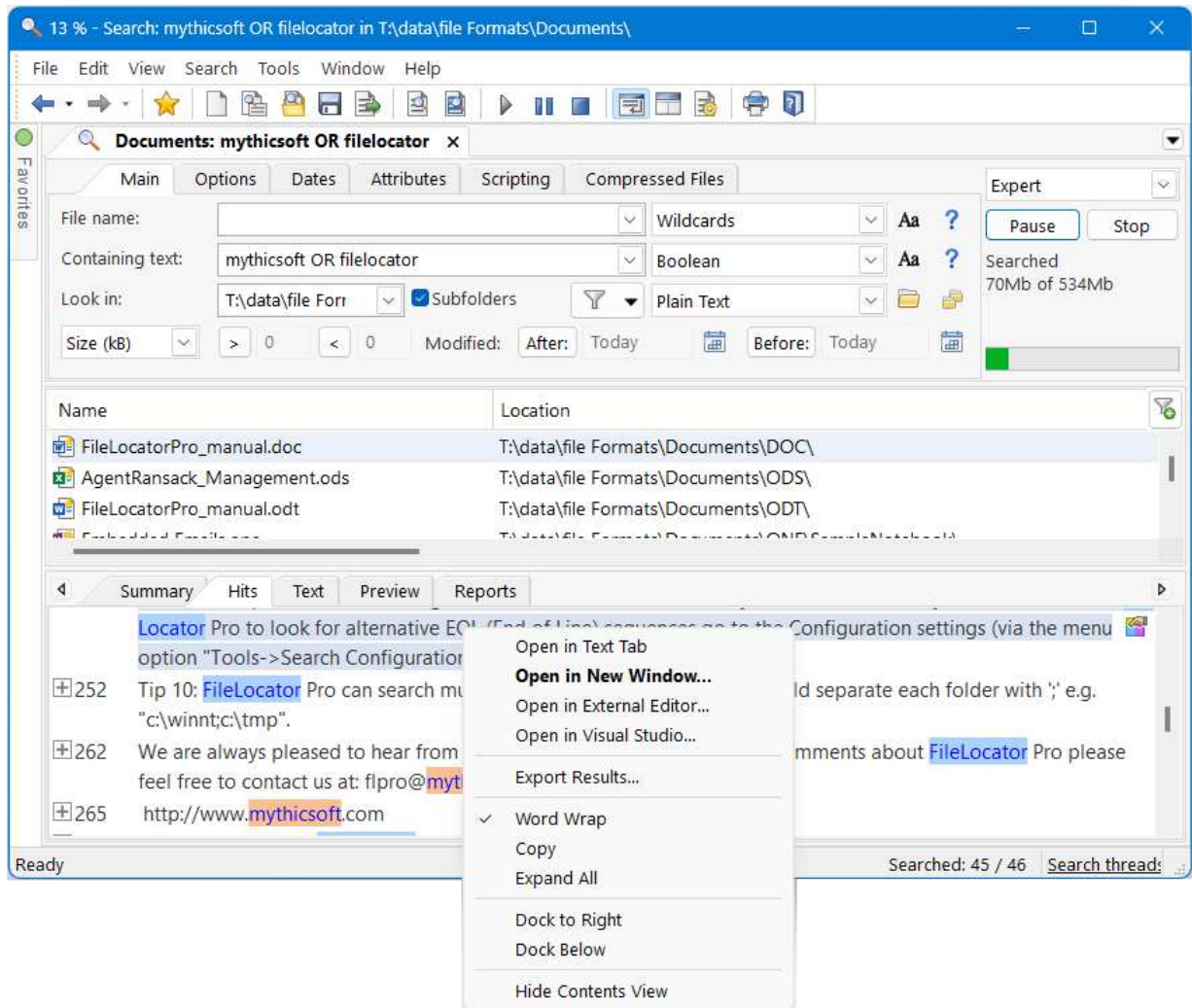


Рисунок 1.3 – Agent Ransack

Його сильний технічний бік – здатність виконувати складний багатопотоковий пошук не лише за іменами, а й всередині текстових документів, логів чи вихідних файлів коду. Програма повноцінно підтримує апарат регулярних виразів (Regular Expressions). Завдяки цьому системний адміністратор може написати специфічний шаблон і, наприклад, за один прохід знайти всі IP-адреси, номери помилок або конкретні фрагменти коду в масиві лог-файлів обсягом у кілька гігабайтів [10].

Однак саме через таку вузьку технічну спрямованість Agent Ransack має дуже помітне слабе місце – перевантажений користувацький інтерфейс. Робоче вікно програми буквально всіяне різноманітними вкладками, текстовими полями, випадаючими списками та неочевидними перемикачами. Для того щоб вирішити найпростішу щоденну задачу (наприклад, знайти загублений PDF-документ у папці із завантаженнями), людина змушена візуально відфільтровувати десятки опцій, які їй зараз абсолютно не потрібні.

З точки зору сучасних правил проєктування інтерфейсів, такий підхід суттєво підвищує когнітивне навантаження [11]. Людина не повинна витрачати час, намагаючись зрозуміти, у яке саме поле вводити запит, або згадувати синтаксис регулярних виразів для банального пошуку відеофайлів. Сучасний інтерфейс повинен приховувати складну інженерну логіку «під капотом», пропонуючи натомість зрозумілі візуальні елементи – календар для вибору дат, повзунки для розмірів або звичайні галочки для категорій файлів.

1.3 Результат огляду програмного забезпечення для швидкого пошуку файлів

Проведений аналіз дозволяє зробити висновок, що сучасний ринок утиліт для локального пошуку файлів є доволі сегментованим. Кожен із розглянутих інструментів створювався для вирішення специфічного кола завдань і спирається на власні архітектурні компроміси. Як наслідок, на сьогодні не існує єдиного універсального рішення, і користувач постійно змушений обирати між швидкодією, гнучкістю системи та зручністю інтерфейсу.

Вбудований інструментарій Windows Search пропонує інтеграцію з операційною системою, проте його архітектура, заснована на постійному фоновому індексуванні, виявилася неефективною для сучасних обсягів даних. Пошук у неіндексованих директоріях залишається критично повільним і часто призводить до зависання Провідника, що є неприпустимим під час інтенсивної роботи.

Альтернативні рішення, такі як Everything, успішно долають проблему швидкості. Зчитуючи безпосередньо таблицю MFT, вони забезпечують миттєвий результат. Але цей підхід має суворі технічні обмеження: програма втрачає свою ефективність на мережевих або FAT32-накопичувачах і вимагає підвищених привілеїв адміністратора. Більше того, відсутність інтуїтивних візуальних фільтрів перетворює складний пошук на процес написання специфічних текстових команд, що суперечить сучасним стандартам проєктування користувацького досвіду.

З іншого боку, спеціалізовані утиліти на кшталт Agent Ransack надають вичерпний набір технічних можливостей, зокрема повноцінну підтримку регулярних виразів. Проте їхній перевантажений десятками налаштувань інтерфейс створює високий когнітивний бар'єр. Для виконання рутинної задачі – наприклад, знайти всі архіви певного розміру за минулий тиждень – такий інтерфейс виглядає надмірним.

Отже, виникає чітка практична потреба у розробці збалансованого програмного рішення. Оптимальний інструмент має перебувати посередині між цими крайнощами. З технічної точки зору, програма повинна відмовитися від створення важких постійних баз даних на жорсткому диску. Натомість доцільно використати сучасні асинхронні алгоритми сканування з паралельним завантаженням метаданих в оперативну пам'ять (in-memory індексація). Це дозволить досягти високої швидкодії без жорсткої прив'язки до NTFS і вимог запуску від імені адміністратора.

З точки зору інтерфейсу, нова система має приховувати технічну складність «під капотом». Замість вивчення мови пошукових запитів, користувач повинен отримати набір зрозумілих візуальних інструментів. Саме така комбінація програмної продуктивності та сучасного мінімалістичного дизайну є головним аргументом для створення власного додатку в рамках поточної роботи.

2 ПРОЕКТУВАННЯ ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ДЛЯ ІНДЕКСАЦІЇ ЖОРСТКОГО ДИСКУ ТА ШВИДКОГО ПОШУКУ ФАЙЛІВ

2.1 Постановка цілей та формування вимог до програмних засобів для індексації жорсткого диску та швидкого пошуку файлів

Перш ніж переходити до побудови архітектури та написання коду, потрібно чітко сформулювати цілі проєкту. Спираючись на недоліки існуючих програм, які було розібрано у попередньому розділі, можна стверджувати, що головною метою є розробка десктопного додатку, здатного шукати файли з високою швидкістю, не перевантажуючи при цьому жорсткий диск запущеними фоновими службами. Водночас програма має бути простою у використанні завдяки зрозумілій системі візуальних фільтрів.

Для досягнення цієї мети необхідно деталізувати технічне завдання. У практиці інженерії програмного забезпечення такі задачі прийнято розділяти на дві категорії: функціональні (описують, що саме має робити система) та нефункціональні (визначають, як добре вона має працювати в тих чи інших умовах) [1].

До базових функціональних вимог програми належать:

- а) Можливість вказати шлях до потрібної директорії для початку сканування.
- б) Збір та збереження точної інформації про знайдені об'єкти: ім'я файлу, його повний шлях, розмір у байтах та дату останньої зміни.
- в) Реалізація текстового пошуку з опціональним врахуванням реєстру.
- г) Наявність системи гнучкого фільтрування. Програма повинна вміти відсіювати результати за категоріями (документи, зображення, аудіо, відео, архіви, програми), а також підтримувати вибірку файлів, створених у певний проміжок часу або в межах заданого розміру.

д) Забезпечення базових зручностей взаємодії з операційною системою: підтримка технології drag-and-drop для вибору папок та відкриття файлів подвійним кліком прямо з таблиці результатів.

Серед нефункціональних вимог особливу увагу слід приділити продуктивності та стабільності роботи:

а) Чуйність інтерфейсу. Процес обходу десятків тисяч вкладених папок займає певний час. Протягом цього процесу головне вікно програми не повинно зависати. Користувач має бачити смугу прогресу сканування та мати можливість зупинити його натисканням кнопки скасування.

б) Оптимізація використання оперативної пам'яті. Метадані файлів зберігатимуться у вигляді in-memory індексу. Щоб уникнути перевантаження компонента відображення при обробці надвеликих масивів даних, програма має відображати лише першу тисячу знайдених результатів.

в) Стійкість до системних помилок. Операційна система містить безліч захищених директорій та файлів без прав на читання. Натрапивши на такий файл, сканер не повинен аварійно завершувати роботу; він має тихо проігнорувати помилку доступу і продовжити обхід наступних гілок.

Сформульований перелік вимог є відправною точкою для подальшої роботи. Ці обмеження диктують необхідність використання багатошарової архітектури, потокобезпечних колекцій та асинхронних алгоритмів на наступних етапах проектування.

2.2 Архітектура програмних засобів для індексації жорсткого диску та швидкого пошуку файлів

Вимоги до продуктивності та стабільності, сформульовані у попередньому підрозділі, безпосередньо впливають на вибір програмної структури. Якби весь функціонал (сканування диска, фільтрація, оновлення таблиць на екрані) був написаний в одному монолітному класі, будь-яка важка операція неминуче призводила б до зависання графічного інтерфейсу.

Для вирішення цієї проблеми систему спроектовано за принципом багат шаровості з дотриманням принципу єдиної відповідальності. Програмний код розділено на чотири ізольовані рівні, кожен з яких виконує лише свою специфічну задачу:

а) Рівень взаємодії з операційною системою (System Layer). За цей функціонал відповідає клас FileScanner. Його єдина мета – звернутися до файлової системи Windows за вказаним шляхом, обійти дерево каталогів і зібрати метадані про об'єкти (назву, шлях, розмір, дату зміни). Цей клас нічого не знає про те, як ці дані будуть відфільтровані чи показані на екрані, він лише генерує інформацію.

б) Рівень зберігання даних (Data Layer). Представлений класом FileIndex. Він працює як буфер оперативної пам'яті програми. Його головна архітектурна вимога – забезпечення потокобезпечності. Поки рівень ОС у фоновому режимі записує сюди тисячі нових файлів щосекунди, інші рівні повинні мати змогу паралельно зчитувати ці дані для пошуку без виникнення конфліктів пам'яті.

в) Рівень бізнес-логіки (Business Logic). На цьому рівні працює пошуковий рушій – SearchEngine. Він приймає текстові запити та застосовує логіку фільтрів (математичне порівняння дат або розмірів). Ізоляція логіки пошуку гарантує, що в майбутньому розширення системи (наприклад, додавання пошуку за регулярними виразами) не вимагатиме переписування коду сканера чи візуальних форм.

г) Рівень інтерфейсу користувача (UI Layer). Візуальна оболонка MainForm. Цей шар є максимально "тонким" – він не виконує складних розрахунків. Його зона відповідальності полягає лише в тому, щоб зафіксувати натискання кнопок, надіслати команду рушію та відмалювати готову відфільтровану вибірку в таблиці DataGridView.

Для наочного відображення статичних зв'язків між описаними компонентами розроблено відповідну UML-діаграму.

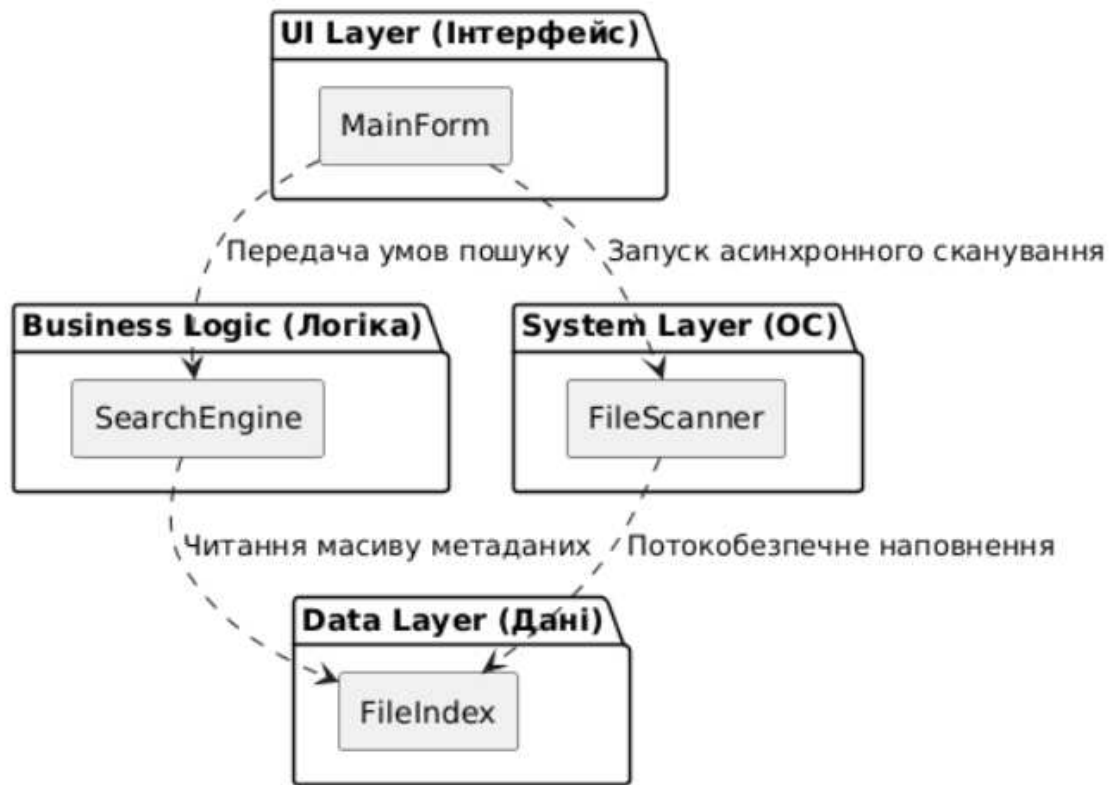


Рисунок 2.1 – Архітектура програмних засобів

Такий структурний розподіл обов'язків гарантує незалежність модулів та запобігає блокуванню користувацького інтерфейсу під час складних дискових операцій. Зафіксувавши статичну структуру класів, можна переходити до розгляду динаміки системи – функціональної схеми, яка описує безпосередній рух потоків даних між цими компонентами.

2.3 Функціональна схема програмних засобів для індексації жорсткого диску та швидкого пошуку файлів

Для розуміння принципів роботи спроектованої системи необхідно розглянути її функціональну схему. Якщо структурна архітектура визначає статичні зв'язки між компонентами, то функціональна модель описує динаміку – логічний потік передачі та обробки даних від моменту ініціалізації пошуку до моменту видачі кінцевого результату.

Роботу розробленого програмного забезпечення можна розділити на два великі ізольовані процеси: процес наповнення бази даних (сканування) та процес вибірки (фільтрація і пошук). Кожен із цих процесів проходить через визначений ланцюг функціональних блоків.

Блок введення та валідації. Це початкова точка взаємодії. Коли користувач обирає директорію для пошуку або змінює параметри фільтрів (дати, розміри, типи файлів), система спочатку перевіряє ці дані на коректність. Наприклад, логіка валідації не дозволить встановити початкову дату більшою за кінцеву, а також автоматично скоригує одиниці виміру, якщо користувач введе некоректний діапазон розмірів (наприклад, від 500 МБ до 10 КБ). Лише після успішної перевірки формується внутрішній об'єкт запиту.

Блок сканування файлової системи. Отримавши валідний шлях до цільової папки, цей функціональний модуль запускає асинхронний обхід дерева каталогів. Його головна задача – згенерувати потік даних. Модуль звертається до операційної системи, читає метадані файлів і відсіює системні папки, до яких немає доступу. Інформація про кожен успішно прочитаний файл пакується у спеціальну структуру даних і передається наступному блоку.

Блок in-memory індексації. Цей модуль отримує неперервний потік даних від сканера і зберігає його в оперативній пам'яті. Він працює як буфер накопичення. Оскільки сканування відбувається у фоновому потоці, цей блок використовує механізми потокобезпечного запису, щоб гарантувати цілісність даних.

Блок пошуку та маршрутизації (Search Engine). Це ядро аналітичної обробки програми. Воно активується щоразу, коли користувач вводить новий символ у рядок пошуку або натискає на візуальний фільтр. Цей блок не звертається до жорсткого диска. Він бере готовий масив даних з in-memory індексу і послідовно пропускає його через каскад математичних та логічних перевірок (порівняння рядків, перевірка входження у діапазон дат, конвертація байтів у мегабайти для порівняння розмірів).

Блок рендерингу результатів. Відфільтрований масив даних передається назад до графічного інтерфейсу. Щоб не перевантажувати підсистему відмальовки вікон операційної системи, цей блок застосовує техніку лімітування (Lazy Loading / Pagination) – виводить на екран лише першу тисячу найрелевантніших записів.

Для наочної демонстрації описаних процесів була розроблена функціональна блок-схема програмного забезпечення.

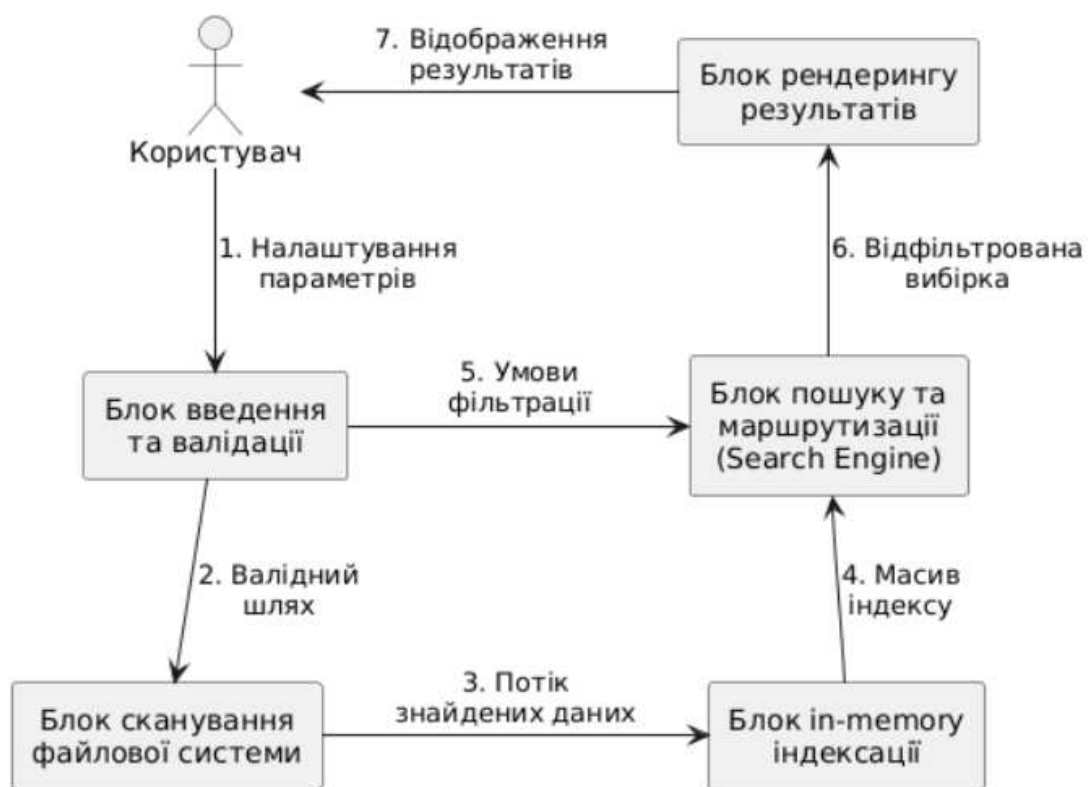


Рисунок 2.2 – Функціональна схема інструментальних засобів для індексації жорсткого диску та швидкого пошуку файлів

Запропонована функціональна модель демонструє однонаправлений потік даних (від жорсткого диска через оперативну пам'ять до екрана користувача). Це гарантує, що важкі дискові операції повністю ізольовані від процесів сортування, що забезпечить стабільну роботу програми навіть під час обробки великих за обсягом директорій.

2.4 Проєктування підсистеми сканування файлової системи

Найнижчим рівнем розробленої архітектури є компонент взаємодії з операційною системою – підсистема сканування. Її пряме завдання полягає у зверненні до вказаної директорії на жорсткому диску, рекурсивному обході всіх вкладених папок та зчитуванні метаданих кожного знайденого файлу.

Під час проєктування цього модуля виникла необхідність вирішити проблему надмірного споживання оперативної пам'яті. Традиційний підхід до читання папок передбачає завантаження повного списку файлів у пам'ять перед початком їх обробки. Для великих дисків із сотнями тисяч об'єктів це створює критичні затримки і перевантажує систему. Тому в розробленій програмі застосовано механізм лінивих обчислень (ітераторів). Додаток отримує дані про кожен файл поштучно, одразу в момент його виявлення на диску. Це дозволяє миттєво передавати інформацію до блоку індексації та без затримок оновлювати візуальну смугу прогресу [12].

Оскільки процес обходу розгалужених файлових дерев є ресурсомістким, його винесено в окремий фоновий потік за допомогою засобів асинхронного програмування. Завдяки цьому головний потік, який відповідає за відображення графічного інтерфейсу, залишається вільним, що запобігає "зависанню" вікна програми. Додатково така асинхронна модель дозволила реалізувати механізм безпечного скасування. Якщо користувач обрав неправильну папку, він може перервати процес у будь-який момент. Сканування зупиниться коректно, миттєво звільнивши зайняті системні ресурси.

Окрему увагу приділено відмовостійкості компонента. Операційні системи сімейства Windows містять значну кількість прихованих та захищених системних директорій. Спроба прямого доступу до них викликає помилку системи безпеки, що зазвичай призводить до аварійної зупинки алгоритму обходу. Для уникнення цієї проблеми реалізовано логіку безпечного сканування. Вона автоматично перехоплює помилки прав доступу або винятки, пов'язані із занадто довгими шляхами до файлів. Алгоритм просто

фіксує неможливість доступу, тихо ігнорує проблемну гілку і продовжує сканування доступних каталогів [2].

Таке архітектурне рішення гарантує стабільну роботу додатка й дозволяє зібрати максимально повний індекс навіть на комп'ютерах із суворими політиками безпеки.

2.5 Організація in-memory індексу для збереження метаданих

Наступним етапом проєктування є організація рівня зберігання даних. Оскільки головною вимогою до програми є забезпечення миттєвої реакції на дії користувача, класичний підхід зі збереженням бази даних на жорсткому диску був відкинутий. Натомість реалізовано концепцію in-memory індексації – механізму, який тримає всю необхідну для фільтрації інформацію безпосередньо в оперативній пам'яті комп'ютера.

Базовою одиницею зберігання на цьому рівні виступає спеціалізована модель даних. Вона являє собою оптимізовану структуру, яка інкапсулює вичерпний набір метаданих про конкретний об'єкт: його назву, повний шлях, розмір, дату останньої зміни та розширення. Завантаження виключно цих атрибутів, без читання фізичного вмісту самого файлу, дозволяє ефективно використовувати ресурси пам'яті навіть під час опрацювання каталогів із сотнями тисяч документів.

Управління масивом цих записів покладено на окремий модуль індексації. Головним інженерним викликом під час його проєктування стала необхідність вирішення проблеми конкурентного доступу (race condition). Специфіка роботи додатку полягає в тому, що підсистема сканування працює у фоновому потоці, безперервно генеруючи та додаючи нові файли до індексу. У той самий час пошуковий рушій повинен мати змогу паралельно зчитувати цей масив для застосування фільтрів, обраних користувачем. Використання стандартних базових колекцій у такому середовищі є неможливим, оскільки спроба одночасної зміни та читання структури різними потоками призведе до помилки виконання програми.

Для надійного збереження масиву метаданих було обрано спеціалізовану потокобезпечну колекцію. Ця структура даних спроектована виключно для сценаріїв паралельного програмування. Вона оптимізована для ситуацій, коли точний порядок додавання елементів не відіграє критичної ролі, але надзвичайно важливою є швидкість та безпека одночасного запису інформації з різних потоків. Такий підхід дозволяє поповнювати базу без використання важких механізмів програмного блокування, які могли б суттєво уповільнити процес фонових сканування.

Ще однією технічною деталлю є механізм підрахунку загальної кількості проіндексованих об'єктів для відображення статистики в інтерфейсі. У багатопотоковому середовищі звичайна математична операція збільшення змінної-лічильника є небезпечною, оскільки два потоки можуть одночасно змінити її значення, що призведе до втрати даних. Щоб розв'язати цю проблему, оновлення лічильника реалізовано за допомогою атомарних операцій. Цей підхід виконує математичну дію на апаратному рівні процесора, гарантуючи точність підрахунку і виключаючи можливість конфлікту між потоками [13].

Завдяки такій організації in-memory індексу програма отримує можливість миттєво віддавати масиви даних пошуковому рушію, зберігаючи при цьому стабільність під час інтенсивних операцій сканування.

2.6 Розробка пошукового рушія та механізмів фільтрації

Логічним центром аналітичної обробки даних у розробленому програмному забезпеченні виступає пошуковий рушій. Його основне завдання – приймати критерії від користувачького інтерфейсу, обробляти їх та повертати відфільтровану вибірку з масиву метаданих, що зберігається в оперативній пам'яті. Важливою архітектурною особливістю цього модуля є повна ізоляція від фізичного жорсткого диска: усі операції порівняння та пошуку виконуються виключно з підготовленим in-memory індексом, що забезпечує миттєву швидкість відгуку.

Процес пошуку реалізовано як каскад логічних та математичних перевірок. Першим етапом є базова текстова фільтрація. Залежно від обраних налаштувань, рушій виконує пошук за точною відповідністю назви об'єкта або з ігноруванням регістру символів. Також розроблено алгоритм для ізольованого пошуку виключно за розширенням, що спрощує індексацію певних форматів.

Другим етапом застосовуються розширені механізми фільтрації, які поділяються на три основні категорії:

а) Фільтрація за типами даних. Рушій містить попередньо визначені списки розширень, згруповані за логічними категоріями (документи, мультимедійні зображення, аудіо, відео, скомпресовані архіви, виконувані програми). Під час обробки масиву система зіставляє розширення кожного запису з обраними категоріями. Якщо користувач не активував жодну з них, цей етап автоматично пропускається для економії обчислювальних ресурсів процесора.

б) Фільтрація за часовим діапазоном. Механізм зіставляє дату останньої модифікації файлу із заданими початковою та кінцевою датами. Для забезпечення високої точності вибірки, час кінцевої межі програмно розширюється до останньої мілісекунди обраної доби. Це гарантує, що абсолютно всі файли, змінені у вказаний день, коректно потраплять до результатів.

в) Фільтрація за розміром. Цей етап вимагає попередньої математичної нормалізації даних. Оскільки інформація про розмір зберігається у внутрішній базі у байтах для максимальної точності, а інтерфейс дозволяє вводити обмеження у похідних одиницях (кілобайтах, мегабайтах, гігабайтах або терабайтах), рушій виконує конвертацію величин. Введені користувачем межі перераховуються у байти за допомогою операцій множення з відповідними коефіцієнтами, після чого відбувається пряме числове порівняння [14].

Такий каскадний підхід до обробки даних дозволяє гнучко комбінувати будь-які пошукові критерії. Система здатна приймати складні багатофакторні запити, послідовно відсіюючи нерелевантні записи та формуючи оптимізований масив результатів для відображення.

2.7 Проєктування користувацького інтерфейсу та взаємодії (UX/UI)

Графічний інтерфейс є точкою контакту між внутрішньою логікою програми та кінцевим користувачем. Найшвидший пошуковий алгоритм втрачає свою цінність, якщо інструменти керування ним незрозумілі або перевантажені зайвими деталями. Тому головним завданням під час проєктування візуальної оболонки було зниження когнітивного навантаження та забезпечення інтуїтивної взаємодії.

Робочий простір головного вікна поділено на кілька чітко розмежованих функціональних зон. Верхня частина відведена під блок маршрутизації та текстового пошуку. Тут користувач може обрати цільову директорію для сканування та ввести пошуковий запит. Бічна або верхня панель містить набір візуальних фільтрів, згрупованих за логікою використання: чекбокси для вибору категорій файлів, календарні елементи для визначення часових рамок та поля для введення розмірних обмежень. Центральну частину вікна займає таблична область, де виводяться результати із зазначенням назви, шляху, точного розміру та дати зміни файлу. Нижня зона відведена під інформаційну панель, яка в режимі реального часу відображає прогрес фонових сканувань або поточний статус системи.

Особливу увагу приділено захисту від помилкового введення даних (концепція Foolproof). Замість того, щоб показувати користувачеві повідомлення про помилки після натискання кнопки пошуку, інтерфейс застосовує превентивну валідацію. Наприклад, якщо користувач випадково обирає початкову дату, яка є пізнішою за кінцеву, програма автоматично коригує значення, прирівнюючи їх.

Схожий алгоритм інтелектуальної корекції розроблено для фільтра розмірів. Якщо ввести мінімальний поріг у 5 ГБ, а максимальний – у 500 МБ, система самостійно розпізнає логічну невідповідність. Вона виконає перерахунок одиниць виміру і автоматично вирівняє межі так, щоб максимальне значення завжди було більшим або дорівнювало мінімальному. Це позбавляє людину необхідності вручну виправляти дрібні помилки введення.

Для прискорення повсякденної роботи в інтерфейс інтегровано кілька сучасних патернів взаємодії. Програма підтримує механізм перетягування об'єктів мишею (Drag-and-Drop). Користувач може просто перетягнути потрібну папку з операційної системи у вікно додатку, і шлях для сканування встановиться автоматично. Відкриття знайденого документа реалізовано звичним подвійним кліком по рядку в таблиці результатів – програма самостійно передасть команду операційній системі на запуск файлу у відповідному редакторі чи програвачі.

Також вирішено проблему візуального "зависання" при виведенні надвеликих масивів даних. Відмальовка сотень тисяч рядків у таблиці вимагає значних ресурсів відеокарти та процесора. Щоб зберегти чуйність інтерфейсу, застосовано підхід лімітованого виведення (Lazy Loading). На екран виводиться лише перша тисяча найрелевантніших записів, а в інформаційній панелі з'являється відповідне повідомлення про застосований ліміт. Практика показує, що людині фізично не потрібно переглядати візуально більше тисячі результатів одночасно – замість цього користувач просто уточнює пошуковий запит.

3 РОЗРОБКА ПРОГРАМНИХ ЗАСОБІВ ДЛЯ ІНДЕКСАЦІЇ ЖОРСТКОГО ДИСКУ ТА ШВИДКОГО ПОШУКУ ФАЙЛІВ

3.1 Алгоритм роботи програмних засобів для індексації жорсткого диску та швидкого пошуку файлів

Розроблений додаток побудований на технології Windows Forms, що передбачає використання подієорієнтованої моделі виконання (event-driven programming) [15]. Це означає, що програма не виконується лінійно від першого до останнього рядка. Натомість вона ініціалізує стартові компоненти і переходить у стан очікування. Подальший рух даних відбувається лише як реакція на конкретні дії користувача – натискання кнопок, перетягування папок або введення тексту.

Глобальний алгоритм життєвого циклу програми розділяється на дві основні гілки виконання: процес наповнення бази даних та процес обробки пошукових запитів. Розглянемо опис алгоритмічних кроків:

а) Ініціалізація. Під час запуску програми створюються екземпляри класів FileIndex (для збереження даних), SearchEngine (для логіки вибірки) та FileScanner (для обходу диска). Також налаштовуються візуальні компоненти, підключаються обробники подій і встановлюються стандартні значення для календарів та одиниць виміру. Після цього програма переходить у стан бездіяльності, не споживаючи ресурсів процесора.

б) Гілка сканування. Коли користувач обирає директорію і запускає пошук, алгоритм спершу валідує шлях. Якщо папки не існує, процес переривається. У разі успіху оперативна пам'ять очищується від попередніх результатів і запускається асинхронний цикл обходу файлового дерева. Алгоритм перевіряє кожен знайдений файл на наявність прав доступу. У разі виникнення помилки доступу спрацьовує гілка ігнорування, яка дозволяє

уникнути аварійного завершення. Коректно зчитані метадані миттєво додаються до потокобезпечного масиву.

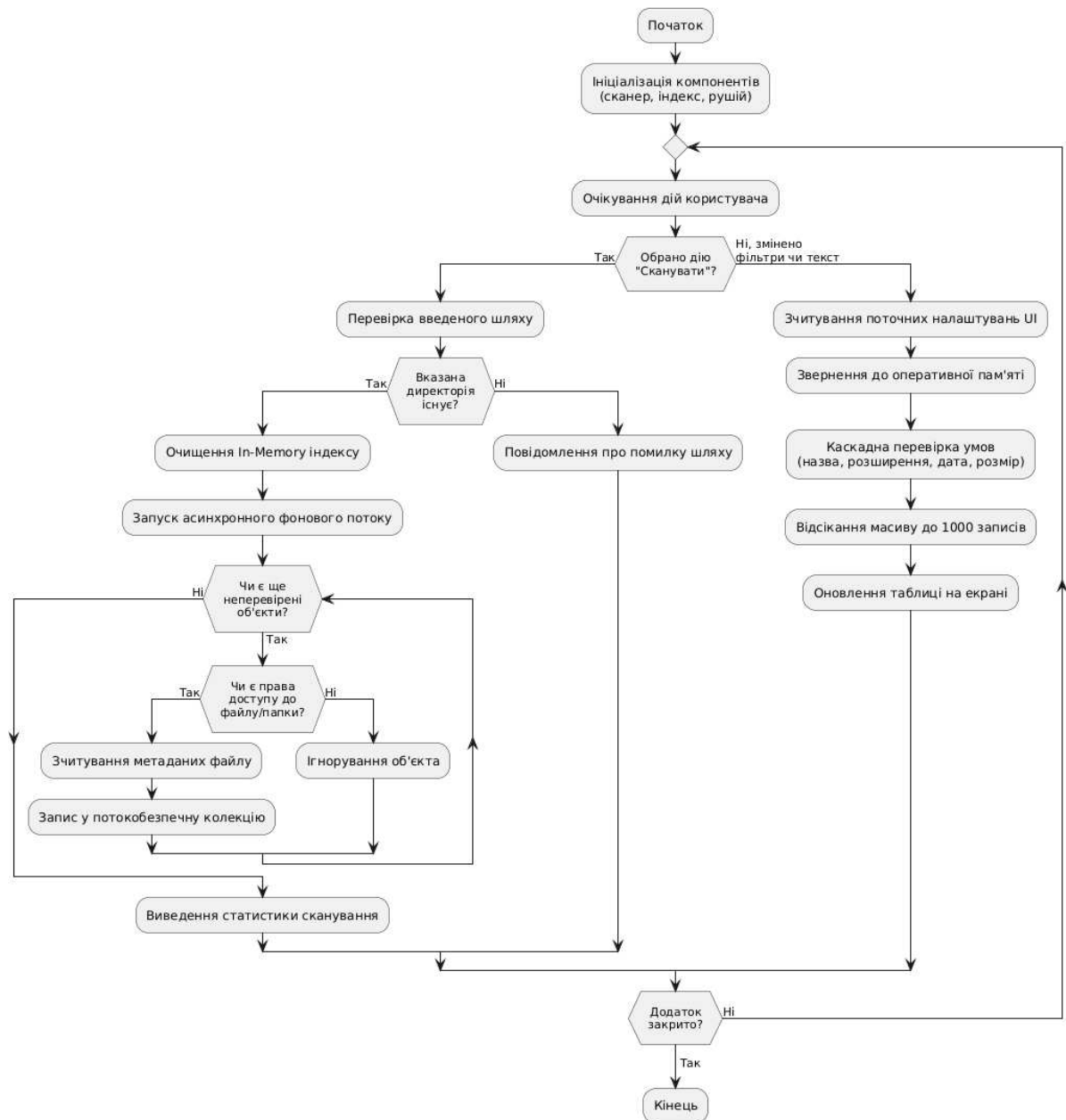


Рисунок 3.1 – Алгоритм роботи програмних засобів для індексації жорсткого диску та швидкого пошуку файлів

в) Гілка пошуку та фільтрації. Ця частина алгоритму запускається автоматично щоразу, коли користувач вводить символ у поле пошуку або ставить галочку біля певного типу файлів (для оптимізації застосовано

невелику затримку через системний таймер, щоб не обробляти кожен символ окремо при швидкому друкуванні). Алгоритм бере масив з оперативної пам'яті і пропускає його через серію послідовних умов: збіг тексту, збіг обраних категорій, входження у часові межі та обмеження розміру.

д) Рендеринг. Відфільтрований результат алгоритмічно обрізається до першої тисячі записів, які потім передаються компоненту таблиці для відображення на екрані.

Така структура алгоритму гарантує, що програма здатна працювати у безперервному циклі, чергуючи важкі процеси читання диска з миттєвими процесами сортування вже завантаженої інформації.

3.2 Реалізація підсистеми асинхронного сканування файлової системи

Основою програмного продукту є підсистема взаємодії з файловою системою операційної системи Windows. За цей функціонал відповідає клас FileScanner. Головною проблемою, яку необхідно було вирішити на етапі кодування, була обробка системних папок, до яких користувач не має прав доступу, а також уникнення блокування графічного інтерфейсу під час тривалого читання жорсткого диска.

Для реалізації безпечного обходу директорій у методі ScanAsync застосовано конструкцію await foreach у поєднанні зі спеціальним ітератором SafeFileEnumerator. У лістингу наведено фрагмент коду, що відповідає за безпосередній обхід файлового дерева.

```
await foreach (var file in
SafeFileEnumerator.EnumerateFilesAsync(
    rootPath, "*", SearchOption.AllDirectories,
    _cancellationTokenSource.Token))
{
    // Перевірка, чи не скасував користувач операцію

    _cancellationTokenSource.Token.ThrowIfCancellationRequested();

    try
    {
```

```

var fileRecord = FileRecord.FromFileInfo(file);
index.Add(fileRecord);
processedCount++;

// Звітуємо про прогрес кожні 100 файлів для уникнення
надмірних оновлень UI
if (processedCount - lastProgressReport >= 100)
{
    progress?.Report(processedCount);
    lastProgressReport = processedCount;
}
}
catch (Exception ex) when (ex is UnauthorizedAccessException
or
                                PathTooLongException or
IOException)
{
    // Пропускаємо окремі файли, до яких немає доступу
    continue;
}
}

```

Як видно з лістингу, алгоритм читає інформацію про файли поштучно. Під час кодування застосовано паттерн зіставлення зі зразком. Це дозволяє алгоритму не "падати" у разі зустрічі з прихованими системними каталогами або надто довгими шляхами, а просто ігнорувати їх за допомогою оператора `continue` і продовжувати роботу.

Ще одним важливим інженерним рішенням стала оптимізація взаємодії фоновому потоку з візуальним інтерфейсом. Оновлення текстового поля та смуги прогресу при кожному знайденому файлі призвело б до перевантаження головного потоку програми. Тому в коді передбачено штучний ліміт: об'єкт *progress* відправляє звіт системі лише кожні 100 оброблених файлів.

Сам запуск цього ресурсомісткого процесу ініціюється з класу головної форми *MainForm* за допомогою інструментів паралельного програмування:

```

_cancellationTokenSource = new CancellationTokenSource();
var progress = new Progress<int>(value =>
{
    _progressBar.Value = Math.Min(value, 100);
    _statusLabel.Text = $"Проскановано {value:N0} файлів...";
});

// Починаємо сканування у фоновому потоці

await Task.Run(async () =>

```

```
{
    await _scanner.ScanAsync(path, _index, progress,
        _cancellationTokenSource.Token);
}, _cancellationTokenSource.Token);
```

Для гарантованого збереження чуйності вікна додатка використовується метод *Task.Run*, який примусово виносить виконання сканера в пул фонових потоків. У свою чергу, клас *Progress<T>* забезпечує безпечну передачу даних про кількість знайдених об'єктів назад до графічного потоку. Також варто відзначити використання *CancellationTokenSource*. Цей об'єкт генерує токен, який передається вглиб методу сканування. Якщо користувач натисне кнопку «Скасувати», токен змінить свій стан, і метод *ThrowIfCancellationRequested()* згенерує виняток, який коректно перерве операцію та звільнить ресурси.

Таким чином, програмна реалізація сканера повністю відповідає раніше висунутим вимогам щодо продуктивності, стабільності та керованості процесом.

3.3 Керівництво користувача та опис роботи програми

Розроблений програмний продукт має інтуїтивно зрозумілий графічний інтерфейс, який забезпечує швидкий доступ до всіх функцій пошуку та фільтрації без необхідності вивчення складних інструкцій. Логіка взаємодії з додатком будується на принципі єдиного вікна (рисунок 3.2) та поділяється на два послідовні етапи: підготовку індексу та безпосередній пошук.

Робота розпочинається з визначення області пошуку – локальної папки чи диска. Вказати цільову директорію можна трьома способами: прямим введенням шляху в адресний рядок, через візуальне діалогове вікно огляду або за допомогою механізму Drag-and-Drop, перетягнувши потрібний каталог безпосередньо у вікно додатка.

Після натискання кнопки «Сканувати» програма розпочинає фоновий збір метаданих. Для запобігання конфліктам доступу інтерфейс тимчасово блокує поле введення шляху, залишаючись при цьому чуйним завдяки асинхронній архітектурі. Процес обходу файлової системи візуалізується за

допомогою динамічної смуги прогресу та інформаційної панелі, що в режимі реального часу відображає кількість знайдених об'єктів.

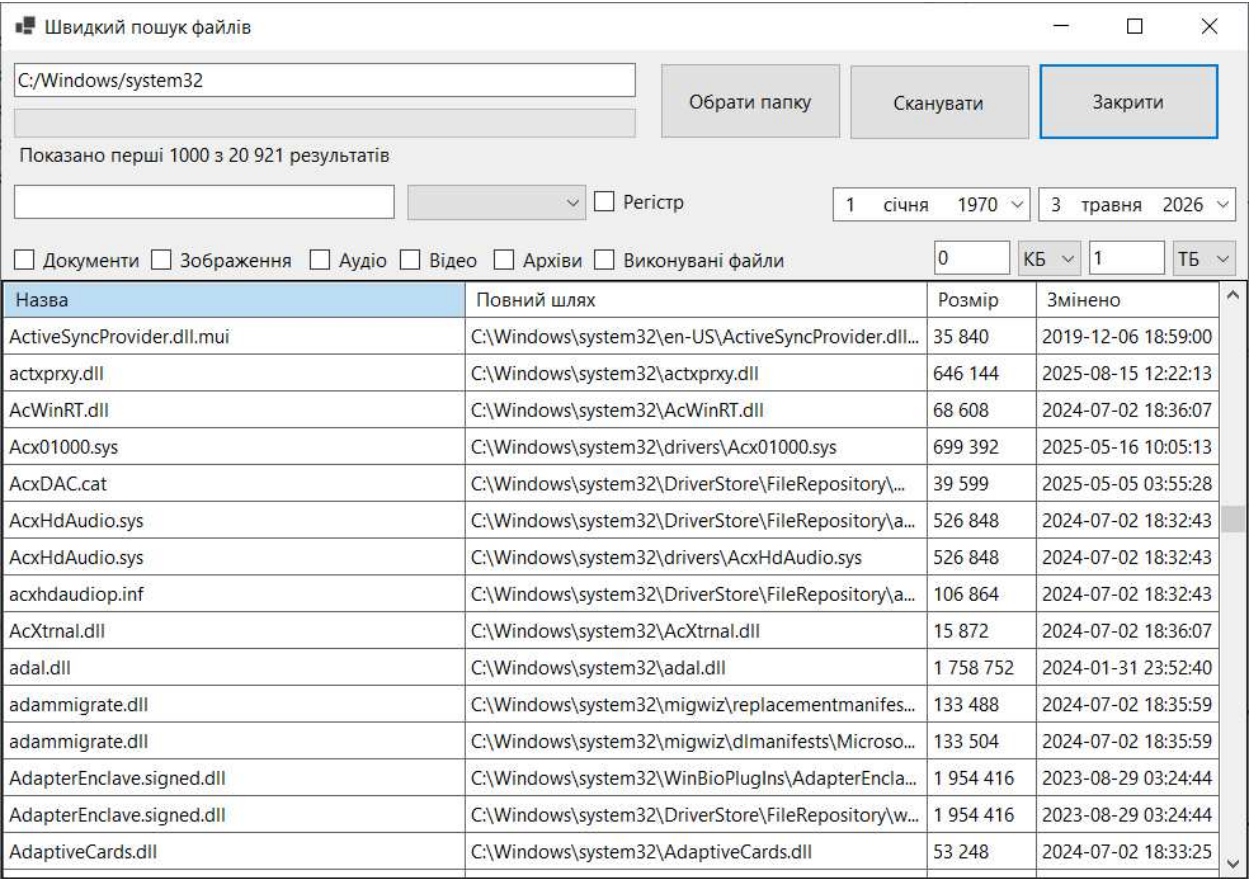


Рисунок 3.2 – Виконання пошуку в заданій директорії без фільтрації

Після успішного завершення індексації всі зібрані метадані знаходяться в оперативній пам'яті, що робить систему готовою до миттєвої обробки запитів. Базовий текстовий пошук ініціюється автоматично одразу під час введення символів у головне поле. За замовчуванням алгоритм шукає часткові збіги в назвах файлів безвідносно до реєстру, проте за потреби користувач може активувати суворий пошук із урахуванням великих та малих літер. Для збереження плавності візуальної оболонки під час швидкого набору тексту застосовано механізм відкладеного виконання. Він оптимізує навантаження на процесор, запускаючи перевірку масиву лише через 300 мілісекунд після натискання останньої клавіші.

Для того, щоб звузити вибірку результатів, передбачено розширений каскад інструментів фільтрації. Усі обрані критерії працюють за принципом послідовного відсіювання (логічне "І"), тому їх можна гнучко комбінувати між собою. За допомогою групи чекбоксів користувач може здійснити категоризацію за типом, миттєво приховавши з екрана все, крім потрібних форматів – документів, зображень, мультимедіа, архівів або програм.

Рисунок 3.3 – Панель фільтрів за ім'ям файлу, регістром та типом документа

Фільтрація за часом модифікації файлу здійснюється через візуальні календарі, оснащені системою автоматичної валідації, яка не дозволить встановити початкову дату пізнішою за кінцеву. Аналогічна превентивна корекція реалізована і для фільтра за обсягом: при заданні меж у зручних похідних одиницях (від кілобайтів до терабайтів) програма самостійно перераховує значення і автоматично вирівнює діапазон, якщо введений мінімальний поріг випадково перевищить максимальний.

Рисунок 3.4 – Панель фільтрів за датою створення файлу і його розміром

Далі відфільтрований масив даних візуалізується у центральній табличній області головного вікна. Для забезпечення інформативності кожен знайдений запис структуровано за чотирма атрибутами: базовою назвою об'єкта, його системним шляхом, обсягом та датою останньої модифікації.

Оскільки відмальовування сотень тисяч рядків є ресурсомісткою операцією, для збереження стабільності графічного інтерфейсу та економії потужностей застосовано механізм лімітованого виведення. Таблиця оптимізована для відображення не більше ніж 1000 записів одночасно, чого цілком достатньо для комфортного сприйняття інформації людиною. У випадку, коли загальна кількість знайдених файлів перевищує цей ліміт, інформаційна панель автоматично змінить свій статус і сповістить про це користувача, пропонуючи застосувати додаткові фільтри для звуження вибірки.

Фінальним кроком взаємодії користувача з програмою є безпосередній доступ до знайденого контенту. Цей процес інтегровано у звичне середовище взаємодії: подвійний клік лівою кнопкою миші по будь-якому рядку таблиці ініціює системний виклик. Додаток передає команду операційній системі Windows, яка здійснює запуск вибраного документа чи медіафайлу в тій програмі, що призначена для обробки цього формату за замовчуванням.

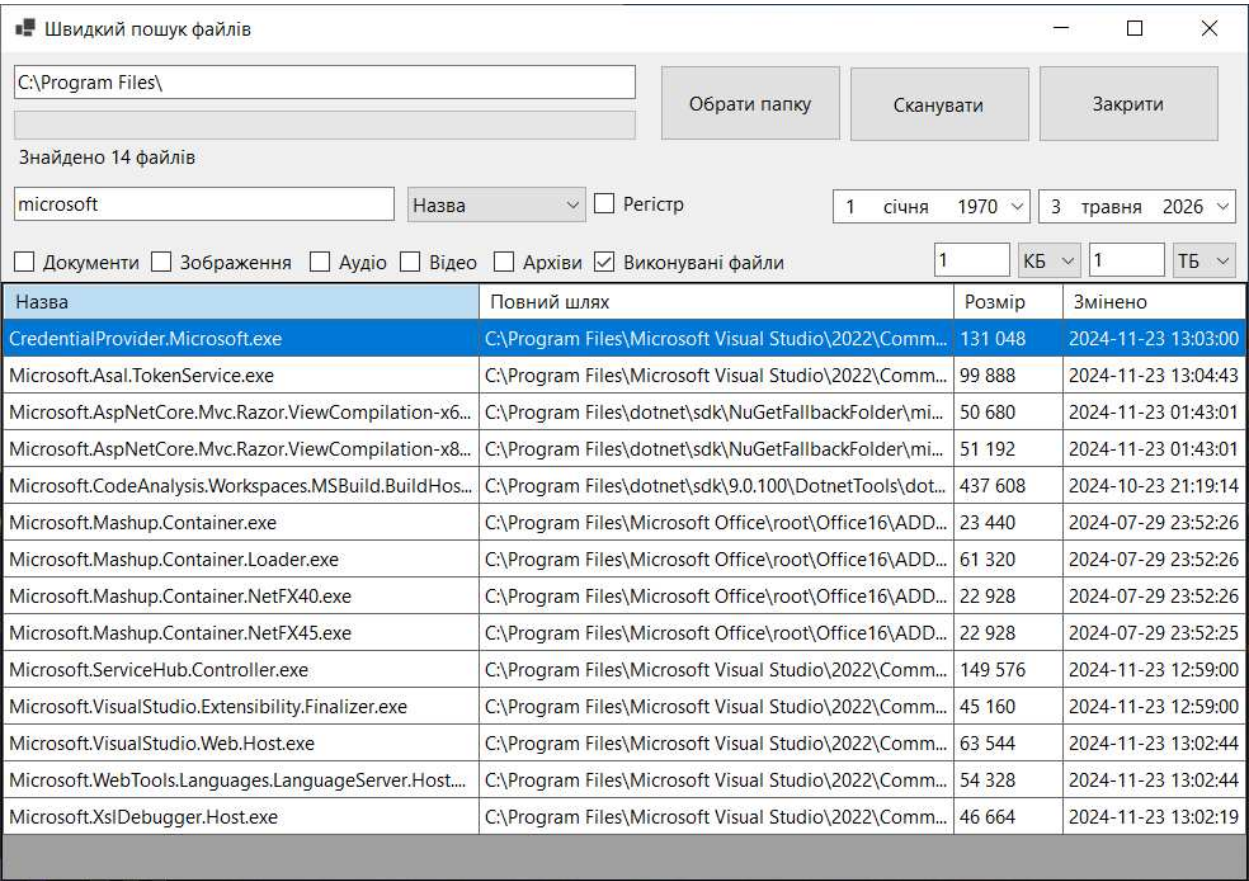


Рисунок 3.5 – Результати пошуку із застосуванням фільтрації

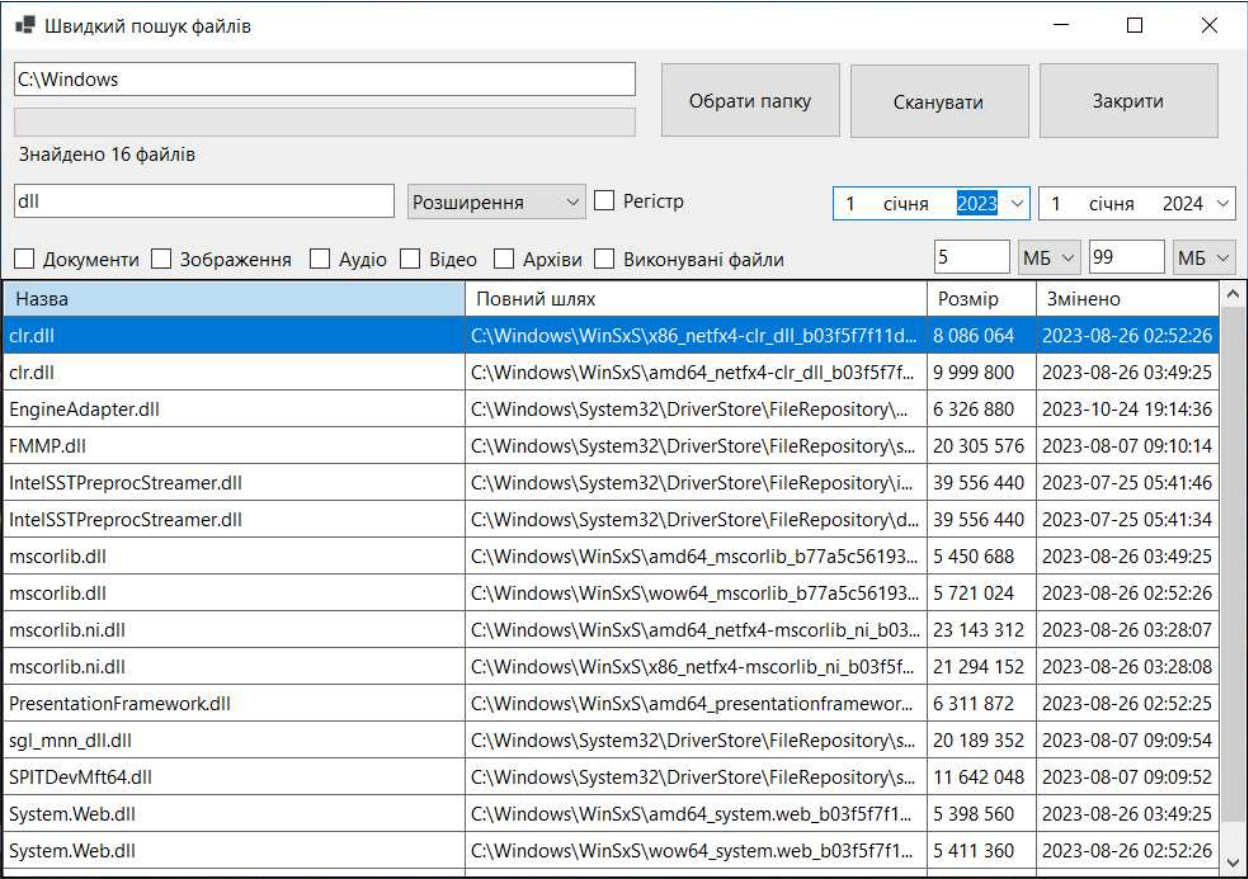


Рисунок 3.6 – Результати пошуку із застосуванням фільтрації

Таким чином, розроблений програмний продукт для індексації жорсткого диску та швидкого пошуку файлів є повністю функціональним і готовим до практичного використання інструментом, що суттєво спрощує навігацію по локальних накопичувачах користувача.

ВИСНОВКИ

В кваліфікаційній роботі було успішно вирішено актуальне завдання – спроектовано та розроблено спеціалізоване програмне забезпечення для швидкого пошуку файлів. Створений продукт ефективно усуває ключові недоліки стандартних пошукових механізмів операційної системи Windows, зокрема їхню низьку швидкість та надмірне навантаження на апаратні ресурси комп'ютера.

Основою досягнення високої швидкодії стала відмова від традиційних реляційних баз даних та прямого читання Master File Table на користь концепції in-memory індексу. Зберігання метаданих безпосередньо в оперативній пам'яті за допомогою потокобезпечних колекцій забезпечило миттєвий доступ до інформації. Процес наповнення цієї бази покладено на розроблений асинхронний сканер файлової системи. Завдяки винесенню алгоритму обходу диска у фоновий потік, графічна оболонка залишається повністю чуйною, а інтеграція механізмів безпечного перехоплення системних винятків гарантує захист від аварійного завершення при неминучих спробах доступу до закритих каталогів операційної системи.

Для обробки зібраних даних спроектовано гнучкий пошуковий рушій із підтримкою каскадної фільтрації, який здатний паралельно аналізувати текстові запити, типи розширень, часові рамки модифікації та обмеження за обсягом. Уся взаємодія з цими алгоритмами відбувається через ергономічний користувацький інтерфейс, який об'єднує сучасні підходи до проєктування: механізм Drag-and-Drop, відкладений запуск пошуку для розвантаження процесора під час швидкого друку та інтелектуальну валідацію параметрів. Додаткова оптимізація споживання обчислювальних ресурсів досягається шляхом лімітованого виведення результатів на екран, що зберігає стабільність таблиці навіть при масивах у сотні тисяч рядків.

У підсумку, розроблений програмний продукт повністю відповідає висунутим на етапі проєктування функціональним та нефункціональним

вимогам. Програма демонструє високу продуктивність, безперебійність роботи та інтуїтивну зрозумілість, що робить її повноцінним, готовим до щоденного використання інструментом. Усі поставлені в дипломній роботі завдання виконано в повному обсязі.

ПЕРЕЛІК ПОСИЛАНЬ

1. Макконнелл С. Досконалий код. Практичний посібник з конструювання програмного забезпечення. Київ : КМ-БУКС, 2019. 896 с.
2. Тролсен Е., Джепікс Ф. Мова програмування C# 9 та платформа .NET 5. Київ : Діалектика, 2021. 1328 с.
3. Норман Д. Дизайн звичних речей. Київ : ArtHuss, 2019. 320 с.
4. Russinovich M., Solomon D., Ionescu A. Windows Internals, Part 1: System architecture, processes, threads, memory management, and more. 7th ed. Redmond : Microsoft Press, 2017. 752 p.
5. Bott E., Siechert C., Stinson C. Windows 10 Inside Out. 3rd ed. Redmond : Microsoft Press, 2020. 1008 p.
6. Everything. URL: <https://www.voidtools.com/en-uk/>
7. Carrier B. File System Forensic Analysis. Boston : Addison-Wesley Professional, 2005. 600 p.
8. Cooper A., Reimann R., Cronin D., Noessel C. About Face: The Essentials of Interaction Design. 4th ed. Indianapolis : Wiley, 2014. 720 p.
9. Agent Ransack. Free File Searching Utility. URL: <http://mythicsoft.com/agentransack/>
10. Goyvaerts J., Levithan S. Regular Expressions Cookbook. 2nd ed. Sebastopol : O'Reilly Media, 2012. 608 p.
11. Krug S. Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability. 3rd ed. Berkeley : New Riders, 2014. 216 p.
12. Albahari J. C# 10 in a Nutshell: The Definitive Reference. Sebastopol : O'Reilly Media, 2022. 1064 p.
13. Тепляков С. Патерни проектування на платформі .NET. Київ : Діалектика, 2018. 320 с.
14. Скит Дж. C# in Depth. 4-те вид. Київ : Діалектика, 2019. 528 с.
15. Event-Driven Programming URL: <https://www.ni.com/docs/en-US/bundle/g-web-development/page/event-driven-programming.html>

Додаток А - Програмний код

MainForm.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;
using FastFileSearch.Models;
using FastFileSearch.Services;

namespace FastFileSearch
{
    /// <summary>
    /// Головна форма програми для швидкого пошуку файлів
    /// </summary>
    public partial class MainForm : Form
    {
        private readonly FileIndex _index;
        private readonly SearchEngine _searchEngine;
        private readonly FileScanner _scanner;
        private CancellationTokenSource?
        _cancellationTokensource;
        private readonly System.Windows.Forms.Timer
        _searchTimer;

        public MainForm()
        {
            _index = new FileIndex();
            _searchEngine = new SearchEngine(_index);
            _scanner = new FileScanner();
            _searchTimer = new System.Windows.Forms.Timer {
Interval = 300 };
            _searchTimer.Tick += SearchTimer_Tick;

            InitializeComponent();

            // Перевіряємо, що DataGridView правильно
            ініціалізований
            SetupDataGridView();
            SetupDragAndDrop();

            // Налаштовуємо валідацію дати
            SetupDateValidation();

            // Налаштовуємо валідацію розмірів
            SetupSizeValidation();
        }
    }
}
```

```

    }

    /// <summary>
    /// Налаштовує валідацію дати для вибору дат
    /// </summary>
    private void SetupDateValidation()
    {
        // Встановлюємо поточну дату для правого елемента
        _toDatePicker.Value = DateTime.Today;

        // Обробник зміни дати для валідації
        _toDatePicker.ValueChanged += (sender, e) =>
        {
            if (_toDatePicker.Value < _fromDatePicker.Value)
            {
                _toDatePicker.Value = _fromDatePicker.Value;
            }
        };

        _fromDatePicker.ValueChanged += (sender, e) =>
        {
            if (_fromDatePicker.Value > _toDatePicker.Value)
            {
                _toDatePicker.Value = _fromDatePicker.Value;
            }
        };
    }

    /// <summary>
    /// Налаштовує валідацію розмірів файлів
    /// </summary>
    private void SetupSizeValidation()
    {
        // Обробник зміни одиниць виміру для валідації
        _minSizeUnitComboBox.SelectedIndexChanged +=
(sender, e) =>
        {
            ValidateSizeRange();
        };

        _maxSizeUnitComboBox.SelectedIndexChanged +=
(sender, e) =>
        {
            ValidateSizeRange();
        };

        // Обробник зміни значень розмірів для валідації
        _minSizeTextBox.TextChanged += (sender, e) =>
        {
            ValidateSizeRange();
        };
    }

```



```

        _maxSizeTextBox.TextChanged += (sender, e) =>
        {
            ValidateSizeRange();
        };
    }

    /// <summary>
    /// Валідує діапазон розмірів файлів
    /// </summary>
    private void ValidateSizeRange()
    {
        if (!string.IsNullOrEmpty(_minSizeTextBox.Text)
        &&
        !string.IsNullOrEmpty(_maxSizeTextBox.Text))
        {
            var minSize = ParseSize(_minSizeTextBox.Text,
            _minSizeUnitComboBox.SelectedItem?.ToString() ?? "KB");
            var maxSize = ParseSize(_maxSizeTextBox.Text,
            _maxSizeUnitComboBox.SelectedItem?.ToString() ?? "TB");

            if (minSize > maxSize)
            {
                // Встановлюємо максимальний розмір рівним
                мінімальному
                var minUnit =
                _minSizeUnitComboBox.SelectedItem?.ToString() ?? "KB";
                var maxUnit =
                _maxSizeUnitComboBox.SelectedItem?.ToString() ?? "TB";

                // Конвертуємо мінімальний розмір в одиниці
                максимального
                var convertedMinSize = minUnit switch
                {
                    "KB" when maxUnit == "MB" => minSize /
                    1024.0,
                    "KB" when maxUnit == "ГБ" => minSize /
                    (1024.0 * 1024),
                    "KB" when maxUnit == "TB" => minSize /
                    (1024.0 * 1024 * 1024),
                    "MB" when maxUnit == "KB" => minSize *
                    1024.0,
                    "MB" when maxUnit == "ГБ" => minSize /
                    1024.0,
                    "MB" when maxUnit == "TB" => minSize /
                    (1024.0 * 1024),
                    "ГБ" when maxUnit == "KB" => minSize *
                    1024.0 * 1024,
                    "ГБ" when maxUnit == "MB" => minSize *
                    1024.0,
                    "ГБ" when maxUnit == "TB" => minSize /
                    1024.0,

```

```

        "TB" when maxUnit == "KB" => minSize *
1024.0 * 1024 * 1024,
        "TB" when maxUnit == "MB" => minSize *
1024.0 * 1024,
        "TB" when maxUnit == "GB" => minSize *
1024.0,
        _ => minSize
    };

    // Встановлюємо значення максимального
розміру рівним мінімальному
    _maxSizeTextBox.Text = Math.Max(1,
Math.Round(convertedMinSize)).ToString();

    // Якщо конвертоване значення занадто велике
для поточної одиниці,
    // змінюємо одиницю на більшу
    if (convertedMinSize > 9999 && maxUnit !=
"TB")
    {
        var nextUnitIndex = Array.IndexOf(new[]
{ "KB", "MB", "GB", "TB" }, maxUnit) + 1;
        if (nextUnitIndex < 4)
        {
            _maxSizeUnitComboBox.SelectedIndex =
nextUnitIndex;

            // Перераховуємо значення для нової
одиниці
            var newUnit =
_maxSizeUnitComboBox.SelectedItem?.ToString() ?? "TB";
            var newConvertedSize = minUnit
switch
            {
                "KB" when newUnit == "MB" =>
minSize / 1024.0,
                "KB" when newUnit == "GB" =>
minSize / (1024.0 * 1024),
                "KB" when newUnit == "TB" =>
minSize / (1024.0 * 1024 * 1024),
                "MB" when newUnit == "KB" =>
minSize * 1024.0,
                "MB" when newUnit == "GB" =>
minSize / 1024.0,
                "MB" when newUnit == "TB" =>
minSize / (1024.0 * 1024),
                "GB" when newUnit == "KB" =>
minSize * 1024.0 * 1024,
                "GB" when newUnit == "MB" =>
minSize * 1024.0,
                "GB" when newUnit == "TB" =>
minSize / 1024.0,
            }
        }
    }

```

```

minSize * 1024.0 * 1024 * 1024,      "TB" when newUnit == "KB" =>
minSize * 1024.0 * 1024,             "TB" when newUnit == "MB" =>
minSize * 1024.0,                     "TB" when newUnit == "TB" =>
_ => minSize
    };
    _maxSizeTextBox.Text = Math.Max(1,
Math.Round(newConvertedSize)).ToString();
    }
    }
    }
}

/// <summary>
/// Налаштовує колонки DataGridView
/// </summary>
private void SetupDataGridView()
{
    _dataGridView.Columns.Clear();

    _dataGridView.Columns.Add(new
DataGridViewTextBoxColumn
    {
        Name = "Name",
        HeaderText = "Назва",
        AutoSizeMode =
DataGridViewAutoSizeMode.Fill
    });

    _dataGridView.Columns.Add(new
DataGridViewTextBoxColumn
    {
        Name = "FullPath",
        HeaderText = "Повний шлях",
        AutoSizeMode =
DataGridViewAutoSizeMode.Fill,
        MinimumWidth = 300
    });

    _dataGridView.Columns.Add(new
DataGridViewTextBoxColumn
    {
        Name = "Size",
        HeaderText = "Розмір",
        Width = 100,
       DefaultCellStyle = new DataGridViewCellStyle {
Format = "N0" }
    });
}

```

```

        _dataGridView.Columns.Add(new
DataGridViewTextBoxColumn
        {
            Name = "Modified",
            HeaderText = "Змінено",
            Width = 150
        });
    }

    /// <summary>
    /// Налаштовує функціональність drag & drop
    /// </summary>
    private void SetupDragAndDrop()
    {
        this.AllowDrop = true;
        this.DragEnter += MainForm_DragEnter;
        this.DragDrop += MainForm_DragDrop;
    }

    /// <summary>
    /// Обробляє натискання кнопки "Сканувати"
    /// </summary>
    private async void ScanButton_Click(object? sender,
EventArgs e)
    {
        var path = _pathTextBox.Text.Trim();

        if (!Directory.Exists(path))
        {
            MessageBox.Show($"Директорію не знайдено:
{path}", "Помилка", MessageBoxButtons.OK, MessageBoxIcon.Error);
            return;
        }

        await StartScanAsync(path);
    }

    /// <summary>
    /// Обробляє натискання кнопки вибору папки
    /// </summary>
    private void BrowseButton_Click(object? sender,
EventArgs e)
    {
        using var folderDialog = new FolderBrowserDialog();
        folderDialog.Description = "Оберіть папку для пошуку
файлів";

        folderDialog.ShowNewFolderButton = true;
        folderDialog.SelectedPath = _pathTextBox.Text;

        if (folderDialog.ShowDialog() == DialogResult.OK)
        {
            _pathTextBox.Text = folderDialog.SelectedPath;

```

```

        }
    }

    /// <summary>
    /// Обробляє натискання кнопки скасування/закриття
    /// </summary>
    private void CancelButton_Click(object? sender,
EventArgs e)
    {
        // Якщо йде процес сканування, скасовуємо його
        if (_cancellationTokenSource != null &&
!_cancellationTokenSource.IsCancellationRequested)
        {
            _cancellationTokenSource.Cancel();
            _statusLabel.Text = "Сканування скасовано";
        }
        else
        {
            // Закриваємо вікно програми
            Close();
        }
    }

    /// <summary>
    /// Починає процес сканування
    /// </summary>
    private async Task StartScanAsync(string path)
    {
        try
        {
            // Оновлюємо стан інтерфейсу
            _scanButton.Enabled = false;
            _cancelButton.Enabled = true;
            _pathTextBox.Enabled = false;
            _progressBar.Value = 0;
            _statusLabel.Text = "Початок сканування...";
            _dataGridView.Rows.Clear();

            _cancellationTokenSource = new
CancellationTokenSource();
            var progress = new Progress<int>(value =>
            {
                _progressBar.Value = Math.Min(value, 100);
                _statusLabel.Text = $"Проскановано
{value:N0} файлів...";
            });

            // Починаємо сканування
            await Task.Run(async () =>
            {
                await _scanner.ScanAsync(path, _index,
progress, _cancellationTokenSource.Token);

```

```

        }, _cancellationTokenSource.Token);

        _statusLabel.Text = $"Сканування завершено.
Знайдено {_index.Count:N0} файлів.";
        RefreshSearchResults();
    }
    catch (OperationCanceledException)
    {
        _statusLabel.Text = "Сканування скасовано.";
    }
    catch (Exception ex)
    {
        _statusLabel.Text = $"Помилка: {ex.Message}";
        MessageBox.Show($"Помилка сканування:
{ex.Message}", "Помилка", MessageBoxButtons.OK,
MessageBoxIcon.Error);
    }
    finally
    {
        // Скидаємо стан інтерфейсу
        _scanButton.Enabled = true;
        _pathTextBox.Enabled = true;
        _progressBar.Value = 0;
        _cancellationTokenSource?.Dispose();
        _cancellationTokenSource = null;
    }
}

/// <summary>
/// Обробляє зміну тексту в полі пошуку
/// </summary>
private void SearchTextBox_TextChanged(object? sender,
EventArgs e)
{
    _searchTimer.Stop();
    _searchTimer.Start();
}

/// <summary>
/// Обробляє спрацювання таймера пошуку для відкладеного
пошуку
/// </summary>
private void SearchTimer_Tick(object? sender, EventArgs
e)
{
    _searchTimer.Stop();
    RefreshSearchResults();
}

/// <summary>
/// Оновлює результати пошуку
/// </summary>

```

```

private void RefreshSearchResults()
{
    try
    {
        // Переконаємось, що DataGridView має колонки
        if (_dataGridView.Columns.Count == 0)
        {
            SetupDataGridView();
        }

        _dataGridView.Rows.Clear();

        var query = _searchTextBox.Text.Trim();
        var filterType =
            _filterComboBox.SelectedItem?.ToString() ?? "Назва";
        var caseSensitive =
            _caseSensitiveCheckBox.Checked;

        List<FileRecord> results;

        if (string.IsNullOrEmpty(query))
        {
            results = _index.GetAll();
        }
        else if (filterType == "Розширення")
        {
            results =
                _searchEngine.SearchByExtension(query);
        }
        else
        {
            results = _searchEngine.SearchByName(query,
caseSensitive);
        }

        // Застосовуємо фільтри за типами файлів
        results = ApplyFileTypeFilters(results);

        // Застосовуємо фільтр за датою створення
        results = ApplyDateFilter(results);

        // Застосовуємо фільтр за розміром файлів
        results = ApplySizeFilter(results);

        foreach (var file in results.Take(1000)) //
Обмежуємо 1000 результатів для продуктивності
        {
            _dataGridView.Rows.Add(
                file.Name,
                file.FullPath,
                file.Size,

```

```

        file.Modified.ToString("yyyy-MM-dd
HH:mm:ss")
    );
}

_statusLabel.Text = results.Count >= 1000
    ? $"Показано перші 1000 з {results.Count:N0}
результатів"
    : $"Знайдено {results.Count:N0} файлів";
}
catch (Exception ex)
{
    _statusLabel.Text = $"Помилка пошуку:
{ex.Message}";
}
}

/// <summary>
/// Застосовує фільтр за розміром файлів
/// </summary>
private List<FileRecord>
ApplySizeFilter(List<FileRecord> files)
{
    // Якщо обидва поля розміру порожні, повертаємо всі
    файли
    if (string.IsNullOrEmpty(_minSizeTextBox.Text)
    &&
        string.IsNullOrEmpty(_maxSizeTextBox.Text))
    {
        return files;
    }

    var filteredFiles = new List<FileRecord>();

    foreach (var file in files)
    {
        var fileSize = file.Size;
        var minSize = ParseSize(_minSizeTextBox.Text,
_minSizeUnitComboBox.SelectedItem?.ToString() ?? "KB");
        var maxSize = ParseSize(_maxSizeTextBox.Text,
_maxSizeUnitComboBox.SelectedItem?.ToString() ?? "MB");

        // Перевіряємо, чи файл входить в діапазон
розмірів
        var withinMinSize =
string.IsNullOrEmpty(_minSizeTextBox.Text) || fileSize >=
minSize;
        var withinMaxSize =
string.IsNullOrEmpty(_maxSizeTextBox.Text) || fileSize <=
maxSize;

        if (withinMinSize && withinMaxSize)

```



```

        {
            filteredFiles.Add(file);
        }
    }

    return filteredFiles;
}

/// <summary>
/// Перетворює розмір файлу з тексту та одиниці в байти
/// </summary>
private long ParseSize(string sizeText, string unit)
{
    if (string.IsNullOrEmpty(sizeText) ||
!long.TryParse(sizeText, out var size))
        return 0;

    return unit switch
    {
        "KB" => size * 1024,
        "MB" => size * 1024 * 1024,
        "GB" => size * 1024 * 1024 * 1024,
        "TB" => size * 1024L * 1024 * 1024 * 1024,
        _ => size
    };
}

/// <summary>
/// Застосовує фільтр за датою створення файлів
/// </summary>
private List<FileRecord>
ApplyDateFilter(List<FileRecord> files)
{
    var fromDate = _fromDatePicker.Value.Date;
    var toDate =
_toDatePicker.Value.Date.AddDays(1).AddTicks(-1); // Кінець дня

    return files.Where(file =>
        file.Modified >= fromDate && file.Modified <=
toDate).ToList();
}

/// <summary>
/// Застосовує фільтри за типами файлів
/// </summary>
private List<FileRecord>
ApplyFileTypeFilters(List<FileRecord> files)
{
    // Якільки жоден фільтр не обраний, повертаємо всі
файли
    if (!_documentsCheckBox.Checked &&
!_imagesCheckBox.Checked &&

```

```

        !_audioCheckBox.Checked &&
        !_videoCheckBox.Checked &&
        !_archivesCheckBox.Checked &&
        !_executablesCheckBox.Checked)
    {
        return files;
    }

    var filteredFiles = new List<FileRecord>();

    foreach (var file in files)
    {
        var extension =
file.Extension.ToLowerInvariant();

        // Документи
        if (_documentsCheckBox.Checked &&
IsDocumentFile(extension))
        {
            filteredFiles.Add(file);
            continue;
        }

        // Зображення
        if (_imagesCheckBox.Checked &&
IsImageFile(extension))
        {
            filteredFiles.Add(file);
            continue;
        }

        // Аудіо
        if (_audioCheckBox.Checked &&
IsAudioFile(extension))
        {
            filteredFiles.Add(file);
            continue;
        }

        // Відео
        if (_videoCheckBox.Checked &&
IsVideoFile(extension))
        {
            filteredFiles.Add(file);
            continue;
        }

        // Архіви
        if (_archivesCheckBox.Checked &&
IsArchiveFile(extension))
        {
            filteredFiles.Add(file);

```

```

        continue;
    }

    // Виконувані файли
    if (_executablesCheckBox.Checked &&
IsExecutableFile(extension))
    {
        filteredFiles.Add(file);
        continue;
    }
}

return filteredFiles;
}

/// <summary>
/// Перевіряє, чи є файл документом
/// </summary>
private bool IsDocumentFile(string extension)
{
    var documentExtensions = new[] { ".pdf", ".doc",
".docx", ".txt", ".rtf", ".odt", ".xls", ".xlsx", ".ppt",
".pptx", ".csv", ".html", ".htm", ".md", ".epub", ".mobi" };
    return documentExtensions.Contains(extension);
}

/// <summary>
/// Перевіряє, чи є файл зображенням
/// </summary>
private bool IsImageFile(string extension)
{
    var imageExtensions = new[] { ".jpg", ".jpeg",
".png", ".gif", ".bmp", ".tiff", ".tif", ".svg", ".ico",
".webp", ".psd", ".raw", ".heic", ".avif" };
    return imageExtensions.Contains(extension);
}

/// <summary>
/// Перевіряє, чи є файл аудіо
/// </summary>
private bool IsAudioFile(string extension)
{
    var audioExtensions = new[] { ".mp3", ".wav",
".flac", ".aac", ".ogg", ".wma", ".m4a", ".opus", ".aiff",
".au", ".ra", ".amr" };
    return audioExtensions.Contains(extension);
}

/// <summary>
/// Перевіряє, чи є файл відео
/// </summary>
private bool IsVideoFile(string extension)

```

```

        {
            var videoExtensions = new[] { ".mp4", ".avi",
            ".mkv", ".mov", ".wmv", ".flv", ".webm", ".m4v", ".3gp", ".mpg",
            ".mpeg", ".ogv", ".ts", ".vob" };
            return videoExtensions.Contains(extension);
        }

        /// <summary>
        /// Перевіряє, чи є файл архівом
        /// </summary>
        private bool IsArchiveFile(string extension)
        {
            var archiveExtensions = new[] { ".zip", ".rar",
            ".7z", ".tar", ".gz", ".bz2", ".xz", ".lzma", ".cab", ".iso",
            ".dmg", ".deb", ".rpm", ".apk" };
            return archiveExtensions.Contains(extension);
        }

        /// <summary>
        /// Перевіряє, чи є файл виконуваним
        /// </summary>
        private bool IsExecutableFile(string extension)
        {
            var executableExtensions = new[] { ".exe", ".msi",
            ".bat", ".cmd", ".com", ".scr", ".app", ".deb", ".rpm", ".dmg",
            ".pkg", ".run", ".bin" };
            return executableExtensions.Contains(extension);
        }

        /// <summary>
        /// Обробляє подвійний клік по DataGridView
        /// </summary>
        private void DataGridView_DoubleClick(object? sender,
        EventArgs e)
        {
            if (_dataGridView.SelectedRows.Count == 0)
                return;

            var row = _dataGridView.SelectedRows[0];
            var fullPath =
            row.Cells["FullPath"].Value?.ToString();

            if (!string.IsNullOrEmpty(fullPath) &&
            File.Exists(fullPath))
            {
                try
                {
                    Process.Start(new ProcessStartInfo
                    {
                        FileName = fullPath,
                        UseShellExecute = true
                    });
                }
            }
        }
    }
}

```

```

        }
        catch (Exception ex)
        {
            MessageBox.Show($"Не вдалося відкрити файл:
{ex.Message}", "Помилка", MessageBoxButtons.OK,
MessageBoxIcon.Error);
        }
    }

    /// <summary>
    /// Обробляє подію drag enter
    /// </summary>
    private void MainForm_DragEnter(object? sender,
DragEventArgs e)
    {
        if (e.Data?.GetDataPresent(DataFormats.FileDrop) ==
true)
        {
            var files =
(string[])e.Data.GetData(DataFormats.FileDrop);
            if (files.Length == 1 &&
Directory.Exists(files[0]))
            {
                e.Effect = DragDropEffects.Copy;
                return;
            }
        }
        e.Effect = DragDropEffects.None;
    }

    /// <summary>
    /// Обробляє подію drag drop
    /// </summary>
    private void MainForm_DragDrop(object? sender,
DragEventArgs e)
    {
        if (e.Data?.GetDataPresent(DataFormats.FileDrop) ==
true)
        {
            var files =
(string[])e.Data.GetData(DataFormats.FileDrop);
            if (files.Length == 1 &&
Directory.Exists(files[0]))
            {
                _pathTextBox.Text = files[0];
                _ = StartScanAsync(files[0]);
            }
        }
    }

    /// <summary>

```

```

        /// Звільняє ресурси
        /// </summary>
        protected override void Dispose(bool disposing)
        {
            if (disposing)
            {
                _searchTimer?.Dispose();
                _cancellationTokenSource?.Dispose();
            }
            base.Dispose(disposing);
        }
    }
}

```

FileIndex.cs

```

using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using FastFileSearch.Models;

namespace FastFileSearch.Services
{
    /// <summary>
    /// Потокобезпечний in-memory індекс файлів
    /// </summary>
    public class FileIndex
    {
        private ConcurrentBag<FileRecord> _files = new();
        private volatile int _count;

        /// <summary>
        /// Повертає кількість файлів в індексі
        /// </summary>
        public int Count => _count;

        /// <summary>
        /// Конструктор
        /// </summary>
        public FileIndex()
        {
            _count = 0;
        }

        /// <summary>
        /// Додає запис про файл до індексу
        /// </summary>
        /// <param name="fileRecord">Запис про файл для
        додавання</param>
        public void Add(FileRecord fileRecord)
        {

```

```

        if (fileRecord != null)
        {
            _files.Add(fileRecord);
            System.Threading.Interlocked.Increment(ref
_count);
        }
    }

    /// <summary>
    /// Очищує всі записи з індексу
    /// </summary>
    public void Clear()
    {
        // Створюємо новий пакет для очищення всіх елементів
        var newBag = new ConcurrentBag<FileRecord>();
        ExchangeField(ref _files, newBag);
        System.Threading.Interlocked.Exchange(ref _count,
0);
    }

    // Допоміжний метод для обміну поля ConcurrentBag
    private static void ExchangeField(ref
ConcurrentBag<FileRecord> field, ConcurrentBag<FileRecord>
newValue)
    {
        field = newValue;
    }

    /// <summary>
    /// Шукає файли за назвою (case-insensitive)
    /// </summary>
    /// <param name="query">Пошуковий запит</param>
    /// <returns>Список відповідних записів про
файли</returns>
    public List<FileRecord> Search(string query)
    {
        if (string.IsNullOrEmpty(query))
            return new List<FileRecord>();

        return _files
            .Where(f => f.Name.Contains(query,
StringComparison.OrdinalIgnoreCase))
            .OrderBy(f => f.Name)
            .ToList();
    }

    /// <summary>
    /// Шукає файли за розширенням
    /// </summary>
    /// <param name="extension">Розширення файлу (з крапкою
або без)</param>

```

```

        /// <returns>Список відповідних записів про
        файли</returns>
        public List<FileRecord> SearchByExtension(string
        extension)
        {
            if (string.IsNullOrEmpty(extension))
                return new List<FileRecord>();

            // Нормалізуємо розширення (переконаємось, що воно
            починається з крапки)
            var normalizedExt = extension.StartsWith(".") ?
            extension : $".{extension}";

            return _files
                .Where(f => f.Extension.Equals(normalizedExt,
                StringComparison.OrdinalIgnoreCase))
                .OrderBy(f => f.Name)
                .ToList();
        }

        /// <summary>
        /// Повертає всі файли в індексі
        /// </summary>
        /// <returns>Усі записи про файли</returns>
        public List<FileRecord> GetAll()
        {
            return _files.OrderBy(f => f.Name).ToList();
        }

        /// <summary>
        /// Повертає файли, змінені в заданому діапазоні дат
        /// </summary>
        /// <param name="startDate">Початкова дата</param>
        /// <param name="endDate">Кінцева дата</param>
        /// <returns>Список відповідних записів про
        файли</returns>
        public List<FileRecord> SearchByDateRange(DateTime
        startDate, DateTime endDate)
        {
            return _files
                .Where(f => f.Modified >= startDate &&
                f.Modified <= endDate)
                .OrderByDescending(f => f.Modified)
                .ToList();
        }

        /// <summary>
        /// Повертає файли в заданому діапазоні розмірів
        /// </summary>
        /// <param name="minSize">Мінімальний розмір в
        байтах</param>

```



```

        /// <param name="maxSize">Максимальний розмір в
байтах</param>
        /// <returns>Список відповідних записів про
файли</returns>
        public List<FileRecord> SearchBySizeRange(long minSize,
long maxSize)
        {
            return _files
                .Where(f => f.Size >= minSize && f.Size <=
maxSize)
                .OrderByDescending(f => f.Size)
                .ToList();
        }
    }
}

```

FileScanner.cs

```

using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;
using FastFileSearch.Models;
using FastFileSearch.Utills;

namespace FastFileSearch.Services
{
    /// <summary>
    /// Сканер файлової системи для побудови індексу файлів
    /// </summary>
    public class FileScanner
    {
        private volatile bool _isScanning;
        private CancellationTokenSource?
_cancellationTokensource;

        /// <summary>
        /// Повертає, чи виконується сканування в даний момент
        /// </summary>
        public bool IsScanning => _isScanning;

        /// <summary>
        /// Асинхронно сканує файлову систему та будує індекс
        /// </summary>
        /// <param name="rootPath">Кореневий шлях для
сканування</param>
        /// <param name="index">Індекс файлів для
заповнення</param>
        /// <param name="progress">Об'єкт для звітування
прогресу</param>
        /// <param name="cancellationTokens">Токен скасування
операції</param>
    }
}

```

```

        /// <returns>Task, що представляє операцію
сканування</returns>
        public async Task ScanAsync(
            string rootPath,
            FileIndex index,
            IProgress<int> progress,
            CancellationToken cancellationToken)
        {
            if (string.IsNullOrEmpty(rootPath))
                throw new ArgumentException("Root path cannot be
null or empty", nameof(rootPath));

            if (index == null)
                throw new ArgumentNullException(nameof(index));

            if (!Directory.Exists(rootPath))
                throw new DirectoryNotFoundException($"Directory
not found: {rootPath}");

            _isScanning = true;
            _cancellationTokenSource =
CancellationTokenSource.CreateLinkedTokenSource(cancellationToker
n);

            try
            {
                // Очищуємо індекс перед початком
index.Clear();

                var processedCount = 0;
                var lastProgressReport = 0;

                // Використовуємо Directory.EnumerateFiles для
ефективної енумерації
                await foreach (var file in
SafeFileEnumerator.EnumerateFilesAsync(
                    rootPath,
                    "*",
                    SearchOption.AllDirectories,
                    _cancellationTokenSource.Token))
                {

                    _cancellationTokenSource.Token.ThrowIfCancellationRequested();

                    try
                    {
                        var fileRecord =
FileRecord.FromFileInfo(file);
                        index.Add(fileRecord);
                        processedCount++;
                    }
                }
            }
        }
    }
}

```

```

        // Звітуємо про прогрес кожні 100 файлів
        для уникнення надмірних оновлень UI
        if (processedCount - lastProgressReport
    >= 100)
        {
            progress?.Report(processedCount);
            lastProgressReport = processedCount;
        }
        catch (Exception ex) when (ex is
UnauthorizedAccessException or PathTooLongException or
IOException)
        {
            // Пропускаємо окремі файли, до яких
    немає доступу
            continue;
        }

        // Фінальний звіт про прогрес
        progress?.Report(processedCount);
    }
    finally
    {
        _isScanning = false;
        _cancellationTokenSource?.Dispose();
        _cancellationTokenSource = null;
    }
}

/// <summary>
/// Скасовує поточну операцію сканування
/// </summary>
public void Cancel()
{
    if (_isScanning && _cancellationTokenSource != null)
    {
        _cancellationTokenSource.Cancel();
    }
}

/// <summary>
/// Повертає орієнтовну кількість файлів у директорії
(для оцінки прогресу)
/// </summary>
/// <param name="rootPath">Кореневий шлях</param>
/// <param name="cancellationToken">Токен
скасування</param>
/// <returns>Орієнтовна кількість файлів</returns>
public async Task<int> EstimateFileCountAsync(string
rootPath, CancellationToken cancellationToken = default)
{

```

```

        if (string.IsNullOrEmpty(rootPath) ||
!Directory.Exists(rootPath))
            return 0;

        var count = 0;

        try
        {
            await foreach (var file in
SafeFileEnumerator.EnumerateFilesAsync(
                rootPath,
                "*",
                SearchOption.AllDirectories,
                cancellationToken))
            {
                cancellationToken.ThrowIfCancellationRequested();
                count++;

                // Обмежуємо оцінку для уникнення довгих
очікувань
                if (count >= 10000)
                    break;
            }
        }
        catch (OperationCanceledException)
        {
            // Повертаємо кількість, яку встигли порахувати
        }

        return count;
    }
}

```

SearchEngine.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using FastFileSearch.Models;

namespace FastFileSearch.Services
{
    /// <summary>
    /// Пошуковий движок для запитів до індексу файлів
    /// </summary>
    public class SearchEngine
    {
        private readonly FileIndex _index;

        /// <summary>

```

```

    /// Конструктор
    /// </summary>
    /// <param name="index">Індекс файлів для пошуку</param>
    public SearchEngine(FileIndex index)
    {
        _index = index ?? throw new
ArgumentNullException(nameof(index));
    }

    /// <summary>
    /// Шукає файли за назвою
    /// </summary>
    public List<FileRecord> SearchByName(string query, bool
caseSensitive = false)
    {
        if (string.IsNullOrEmpty(query))
            return _index.GetAll();

        var comparison = caseSensitive ?
StringComparison.Ordinal : StringComparison.OrdinalIgnoreCase;

        return _index.GetAll()
            .Where(f => f.Name.Contains(query, comparison))
            .OrderBy(f => f.Name)
            .ToList();
    }

    /// <summary>
    /// Шукає файли за розширенням
    /// </summary>
    /// <param name="extension">Розширення файлу</param>
    /// <returns>Список відповідних записів про
файли</returns>
    public List<FileRecord> SearchByExtension(string
extension)
    {
        if (string.IsNullOrEmpty(extension))
            return new List<FileRecord>();

        return _index.SearchByExtension(extension);
    }

    /// <summary>
    /// Розширений пошук з кількома критеріями
    /// </summary>
    /// <param name="query">Запит за назвою
(опціонально)</param>
    /// <param name="extension">Фільтр розширення
(опціонально)</param>
    /// <param name="minSize">Мінімальний розмір в байтах
(опціонально)</param>

```

```

        /// <param name="maxSize">Максимальний розмір в байтах
(опціонально)</param>
        /// <param name="startDate">Фільтр початкової дати
(опціонально)</param>
        /// <param name="endDate">Фільтр кінцевої дати
(опціонально)</param>
        /// <param name="caseSensitive">Чи повинен пошук за
назвою враховувати регістр</param>
        /// <returns>Список відповідних записів про
файли</returns>
        public List<FileRecord> AdvancedSearch(
            string? query = null,
            string? extension = null,
            long? minSize = null,
            long? maxSize = null,
            DateTime? startDate = null,
            DateTime? endDate = null,
            bool caseSensitive = false)
        {
            var results = _index.GetAll();

            // Фільтруємо за назвою
            if (!string.IsNullOrEmpty(query))
            {
                var comparison = caseSensitive ?
StringComparison.Ordinal : StringComparison.OrdinalIgnoreCase;
                results = results.Where(f =>
f.Name.Contains(query, comparison)).ToList();
            }

            // Фільтруємо за розширенням
            if (!string.IsNullOrEmpty(extension))
            {
                var normalizedExt = extension.StartsWith(".") ?
extension : $".{extension}";
                results = results.Where(f =>
f.Extension.Equals(normalizedExt,
StringComparison.OrdinalIgnoreCase)).ToList();
            }

            // Фільтруємо за розміром
            if (minSize.HasValue)
            {
                results = results.Where(f => f.Size >=
minSize.Value).ToList();
            }

            if (maxSize.HasValue)
            {
                results = results.Where(f => f.Size <=
maxSize.Value).ToList();
            }
        }
    }
}

```

```

        // Фільтруємо за датою
        if (startDate.HasValue)
        {
            results = results.Where(f => f.Modified >=
startDate.Value).ToList();
        }

        if (endDate.HasValue)
        {
            results = results.Where(f => f.Modified <=
endDate.Value).ToList();
        }

        return results.OrderBy(f => f.Name).ToList();
    }

    /// <summary>
    /// Повертає загальну кількість файлів в індексі
    /// </summary>
    /// <returns>Загальна кількість файлів</returns>
    public int GetTotalCount()
    {
        return _index.Count;
    }

    /// <summary>
    /// Повертає статистику про проіндексовані файли
    /// </summary>
    /// <returns>Словник зі статистичними даними</returns>
    public Dictionary<string, object> GetStatistics()
    {
        var allFiles = _index.GetAll();

        return new Dictionary<string, object>
        {
            ["TotalFiles"] = allFiles.Count,
            ["TotalSize"] = allFiles.Sum(f => f.Size),
            ["AverageFileSize"] = allFiles.Count > 0 ?
allFiles.Average(f => f.Size) : 0,
            ["LargestFile"] = allFiles.OrderByDescending(f
=> f.Size).FirstOrDefault()?.Name ?? "Немає",
            ["SmallestFile"] = allFiles.OrderBy(f =>
f.Size).FirstOrDefault()?.Name ?? "Немає",
            ["MostCommonExtension"] = allFiles
                .GroupBy(f => f.Extension)
                .OrderByDescending(g => g.Count())
                .FirstOrDefault()?.Key ?? "Немає"
        };
    }
}

```