Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

Оптимізація сканування файлової системи для APFS (Apple File System)

Текстова частина до кваліфікаційної роботи
за спеціальністю “Інженерія програмного забезпечення” - 121

Керівник кваліфікаційної роботи
Старший викладач
Франків О.О.
_____
(Підпис)
“___” _____ 2025 року
Виконав студент БП ІПЗ-4
Левченко А. С.
“___” _____ 2025 року

Київ – 2025

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Старший викладач
_____О. О. Франків
(підпис)
„____”_____2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на кваліфікаційну роботу

студенту 4 р.н. програми бакалаврату “Інженерія програмного забезпечення”
Левченко Артему Сергійовичу
Розробити комп’ютерну систему для оптимізація сканування файлової системи для APFS (Apple File System)

Зміст текстової частини до кваліфікаційної роботи:

Зміст
Анотація
Вступ
1 API для роботи з файловою системою APFS (Apple File System)
2 Методи сканування та багатопоточної обробки файлової ієрархії
3 Архітектура та порівняльний аналіз розробленого рішення
Висновки
Список літератури

Дата видачі „_1_” листопада 2024 р.

Керівник (підпис)

Завдання отримав (підпис)

ГРАФІК ПІДГОТОВКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	10 жовтня 2024			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	10 жовтня 2024 – 2 листопада 2024			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	2 листопада 2024			
4.	Написання розділів роботи	2 листопада 2024 – 01 березня 2025			
5.	Проміжний контроль виконання роботи	01 лютого 2024			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	11 січня 2025 – 29 березня 2025			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	25 січня 2025			
	Розділ 2 (аналітично-дослідницька частина)	01 березня 2025			
	Розділ 3 (проектно-рекомендаційна частина)	29 березня 2025			
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	01 квітня 2025 – 06 травня 2025			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	26 травня 2025			
9.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено 10 жовтня 2024 р.

Науковий керівник Франків Олександр Олександрович

Виконавець кваліфікаційної роботи Левчено Артем Сергійович

ЗМІСТ

АНОТАЦІЯ

Дана робота присвячена дослідженню та оптимізації процесів сканування файлової системи APFS (Apple File System). У роботі розглянуто основні інструменти для доступу до APFS та алгоритмічні стратегії, такі як верхньорівневий підхід, повний обхід файлової системи, інтерактивний обхід та фільтрація за стоп-словами. Також реалізовані підходи для обробки файлової ієрархії, які включають послідовну та паралельну обробку за допомогою Swift Concurrency та Grand Central Dispatch (GCD). Проведено тестування та порівняльний аналіз ефективності різних методів, які враховують такі критерії, як швидкість, точність, масштабованість та використання ресурсів. В рамках даної роботи створено застосунок для сканування файлової системи APFS, який демонструє практичне застосування розглянутих підходів.

ВСТУП

Файлові системи є невід'ємною частиною сучасних операційних систем, забезпечуючи організацію, збереження та доступ до даних. Ефективність роботи файлової системи має вирішальне значення для продуктивності програмного забезпечення, особливо в умовах роботи з великими обсягами інформації. Apple File System (APFS) була представлена 14 червня 2016 року як файлова система наступного покоління для пристроїв Apple [1]. Вона прийшла на зміну Hierarchical File System Plus (HFS+), яка використовувалася з 1998 року [2].

APFS пропонує низку переваг, включаючи підтримку шифрування, швидкі операції з копіюванням і видаленням файлів, а також покращену продуктивність при роботі з великими обсягами даних. Однак, робота з файловою системою, особливо при скануванні та обробці великих ієрархій файлів і директорій, залишається викликом для розробників, які прагнуть забезпечити високу продуктивність та ефективність своїх додатків.

Саме з огляду на все описане вище, темою даної роботи є оптимізація сканування файлової системи для APFS.

Метою даної роботи є розробка ефективного рішення для сканування файлової системи APFS, яке дозволить дослідити різні підходи та визначити серед застосованих методів сканування найбільш ефективний, який забезпечить мінімізацію часу обробки та оптимізацію використання системних ресурсів. Для досягнення цієї мети буде проведено порівняльний аналіз різних підходів, оцінено їх ефективність та запропоновано можливі шляхи вдосконалення.

Робота складається з трьох основних розділів. У першому розділі розглянуто API для роботи з файловою системою APFS, включаючи використання NSFileManager та URLResourceKey у мові програмування Swift. Також у даному розділі проведено порівняльний аналіз існуючих інструментів.

Другий розділ присвячено розробці та аналізу алгоритмічних стратегій обходу файлової ієрархії. Також детально описано підходи до послідовної та багатопоточної обробки файлової ієрархії. Ці методи дозволяють прискорити

процес сканування за рахунок паралельного виконання задач, що є особливо важливим при роботі з великими обсягами даних.

У третьому розділі представлено архітектуру розробленого рішення. Крім того, проведено порівняльний аналіз ефективності різних методів, що дозволяє оцінити доцільність їхнього використання для сканування файлової системи. На основі результатів тестувань представлено оцінку продуктивності розробленого рішення, включаючи час виконання, використання ресурсів системи та загальну ефективність.

Розділ 1. API для роботи з файловою системою APFS (Apple File System)

1.1 Огляд файлової системи APFS

Apple File System - це нова сучасна файлова система для iOS, macOS, tvOS та watchOS. APFS замінює HFS+ як файлову систему за замовчуванням в iOS 10.3 і новіших версіях, а також в macOS High Sierra і новіших версіях.

APFS являє собою значну технічну еволюцію, спрямовану на кращу обробку вимог сучасних пристроїв Apple. Ця файлова система пропонує покращену швидкість читання і запису файлів, а також кращий розподіл простору.

Багато файлових систем, зокрема HFS+, підтримують лише один том на розділ. Оскільки вільний простір не можна розподіляти між розділами, розмір кожного тому встановлюється під час розбиття сховища пристрою на розділи, і кожен том може збільшуватися лише в межах доступного вільного простору. На противагу цьому, у Apple File System представлено функцію спільного використання простору, яка дозволяє декільком томам в одному контейнері динамічно обмінюватися сховищем [3]. Замість того, щоб виділяти фіксований простір для кожного тома, спільне використання простору дозволяє томам збільшуватися або зменшуватися залежно від потреби, забезпечуючи ефективне використання доступного дискового простору без необхідності переформатування (Рис 1.1).

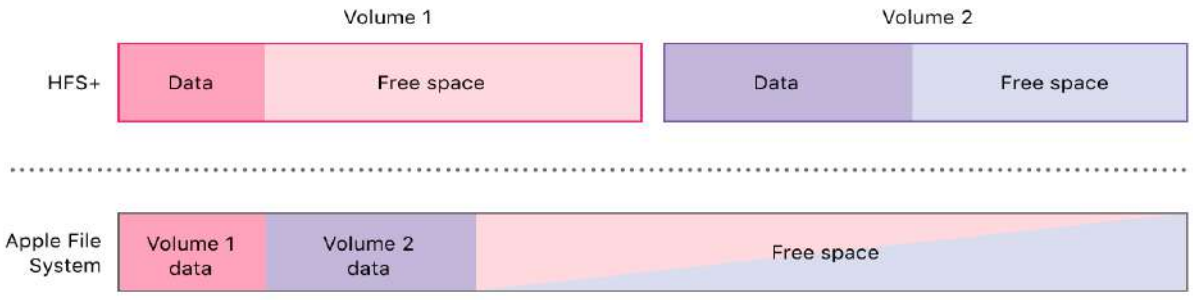

Рис 1.1 Розподіл простору між томами в HFS+ та APFS

Однією з ключових особливостей APFS є те, що вона може створювати знімки, доступні лише для читання, які фіксують стан файлової системи в певний момент часу. Ці знімки дозволяють користувачам швидко відновити файли або повернути систему до попереднього стану без використання додаткового місця для дублікатів даних. Також ця файлова система може миттєво клонувати файли без дублювання даних. Це дозволяє користувачам створювати редаговані копії файлів без збільшення обсягу пам'яті, що робить цю функцію цінною для швидкого резервного копіювання та керування версіями [4].

Також Apple File System забезпечує високу надійність даних завдяки використанню транзакційної моделі, яка допомагає захистити цілісність даних, групуючи оновлення файлової системи в транзакції. У разі збою або раптового вимкнення живлення система може відкотитися до останнього узгодженого стану, чим зменшує ризик пошкодження даних. Крім того, механізм copy-on-write гарантує, що дані не перезаписуються, поки зміни не будуть повністю завершені. Це мінімізує потенційну втрату даних та підвищує загальну стабільність файлової системи.

Безпека даних у APFS підсилюється підтримкою шифрування на рівні файлів. Цей підхід надає користувачам гнучкі можливості керування рівнями безпеки та конфіденційності, і дозволяє окремо захищати як самі файли, так і їхні метадані.

APFS використовує високооптимізовану структуру B-дерева для керування файлами та каталогами, що підвищує продуктивність операцій з каталогами, таких як копіювання, видалення та перейменування файлів. Це призводить до швидшого доступу до файлів, особливо при роботі з каталогами, які містять велику кількість файлів.

У даному досліджені основна увага зосереджена на визначені розміру файлової ієрархії. Для кожного елемента Apple File System визначає два розміри: логічний та фізичний розмір. Логічний розмір (actual size) – це фактичний розмір даних, які містяться у файлі або директорії. Фізичний розмір (allocated size or size

on disk) – це кількість простору, який виділила файлова система для зберігання цього елемента [5]. В даному досліджені ми будемо використовувати фізичний розмір для сканування файлової системи APFS, оскільки він дозволить нам отримати повну інформацію про файлову ієрархію.

Для ефективного доступу, обробки та аналізу даних у Apple File System існують високорівневі API, які спрощують взаємодію, і низькорівневі механізми, які надають більше контролю. У наступних розділах буде розглянуто різні підходи до роботи з APFS. Це дозволить протестувати різні рішення і визначити найкращі підходи для швидкого і надійного сканування даної файлової системи.

1.2 Робота з NSFileManager у Swift

NSFileManager є одним з основних інтерфейсів для доступу до вмісту файлової системи та засобом взаємодії з нею. Клас FileManager надає зручний доступ до спільного об'єкта файлового менеджера, який підходить для більшості типів маніпуляцій з файлами.

Об'єкт FileManager зазвичай є основним способом взаємодії з файловою системою. Він використовується для пошуку, створення, копіювання та переміщення файлів і каталогів. За його допомогою також можна отримати інформацію про файл або каталог, або змінити деякі його атрибути.

За допомогою NSFileManager можна отримати атрибути файлів, які надають детальну інформацію про їхні властивості, такі як розмір, дата створення та інші метадані. Ця інформація є важливою для ефективного управління файловою системою, особливо при роботі з великими обсягами даних. У контексті даного дослідження особливу увага зосереджена на розмірі файлу, який є основним атрибутом для аналізу продуктивності сканування файлової системи APFS. Атрибути файлів можна отримати за допомогою методу attributesOfItem(atPath:) класу NSFileManager. Розмір файлу зберігається під ключем .size і представляється у байтах [6].

NSFileManager дозволяє ефективно отримувати розмір окремих файлів, проте його можливості для роботи з директоріями є обмеженими. Він не надає вбудованого механізму для визначення загального розміру папки, тому її вміст потрібно сканувати рекурсивно. Це робить процес отримання розміру директорії доволі ресурсоємним і часозатратним, особливо у випадку великих файлових структур. Це створює значний виклик для створення ефективного рішення для сканування Apple File System за допомогою NSFileManager.

1.3 Використання URLResourceKey

У сучасній розробці для macOS та IOS для роботи з файловою системою все частіше використовуються об’єкти типу URL замість рядкових шляхів. Це пов’язано з тим, що URL надає більш зручний та безпечний спосіб представлення елементів файлової системи. Також це дозволяє використовувати додаткові можливості такі, як URLResourceKey. URLResourceKey - це набір ключів, які дозволяють отримувати різну інформацію про файли та директорії [7].

Використання URL та URLResourceKey дозволяє уникати помилок пов’язаних з неправильним форматом шляхів до елементів файлової системи та забезпечує більш зручну роботу з файловою системою. Також це дозволяє ефективніше доступитися до даних директорії чи файлу, оскільки опрацьовуються лише необхідні атрибути. Тому це зменшує час на обробку даних.

URLResourceKey надає доступ до широко спектру атрибутів директорій, файлів і навіть томів Apple File System. Однак, як і NSFileManager, URLResourceKey не має окремого атрибуту для розміру директорії, щоб значно спростило сканування файлової системи. Тому у контексті даного дослідження ключову роль відіграють два атрибути: fileSizeKey та fileAllocatedSizeKey. Значення цих ключів можуть суттєво відрізнятись, оскільки вони відображають різні аспекти розміру файлу. Розглянемо дані атрибути детальніше.

Ключ fileSizeKey відображає логічний розмір файлу, тобто фактичний обсяг даних, які зберігаються у файлі. Цей атрибут відображає кількість байтів, які займає файл без урахування додаткових витрат файлової системи APFS. Однак ключ fileAllocatedSizeKey показує фізичний розмір файлу, тобто обсяг дискового простору, який фактично виділено для зберігання цього файлу. Цей розмір може відрізнятися від логічного розміру через особливості роботи Apple File System про які згадувалося вище.

Для сканування файлової системи в даному досліджені основний атрибут є fileAllocatedSizeKey, оскільки fileSizeKey не відображає повноцінний обсяг дискового простору, який займає файл у файловій системі. Окрім цього, сканування Apple File System за допомогою URLResourceKey передбачає рекурсивний обхід файлової ієрархії, аналогічно до підходу з використанням NSFileManager. Проте процес значно оптимізується, бо система оперує лише необхідними атрибутами. Це значно зменшує час обробки великих каталогів і підвищує загальну продуктивність аналізу файлової системи.

1.4 Обчислення розміру директорій за допомогою термінальної команди ‘du’

Ще одним важливим інструментом для сканування Apple File System є термінальна команда du (“disk usage”). Це стандартна утиліта командного рядка Unix/Linux систем, яка використовується для перевірки та оцінки простору, зайнятого файлами і каталогами у файловій системі [8].

Команда du підтримує велику кількість параметрів, які дозволяють налаштувати вивід інформації відповідно до потреб користувача. В контексті даного дослідження найбільш важливими є такі параметри:

- -a (all) - відображає розмір для кожного файла у файловій ієрархії
- -s (summarize) - відображає загальний розмір файлу або директорії без деталізації по піддиректоріях
- -d (depth) - відображає розмір для всіх файлів та директорій лише на вказану глибину

Для інтеграції команди du в додаток використовується клас Process. Він допомагає запустити іншу програму як підпроцес і контролювати її виконання. Також потрібно вказати розташування виконуваного файлу /usr/bin/du та вказати необхідні параметри. Під час виконання команди du ініціалізується новий процес, який виконується незалежно від основного потоку [9]. Після завершення даного процесу потрібно розбити вивід на окремі компоненти, видалити зайві пробіли та обробити дані для подальшого аналізу. Також команда du відображає не розмір файла у байтах, а кількість блоків, які використовуються даним файлом. Якщо елемент файлової системи є директорією, то кількість блоків, про які повідомляється, є сумою блоків, виділених для файлів у директорії, і блоків, виділених для самої директорії. Тому для отримання розміру у байтах потрібно помножити кількість блоків на розмір одного блоку у файловій системі APFS, який становить 512 байтів.

Команда du має досить багато переваг. Дана команда має високу швидкість виконання, бо вона оптимізована для роботи з файловою системою на рівні ядра. Також немає необхідності рекурсивного обходу директорій, що спрощує реалізацію та пришвидшує сканування системи.

Проте існують і певні недоліки даного інструменту. Одним з викликів є відсутність доступу до певних директорій, тому потрібно коректно обробляти ці виняткові ситуації. Крім того, вивід може бути оброблений лише після повного завершення процесу сканування за допомогою команди du. Це унеможливлює поступову обробку даних в процесі виконання команди, тому час сканування може бути більший порівняно з іншими API.

Отже, команда du є досить ефективним інструментом для сканування Apple File System. Вона дозволяє отримати точні дані використаного простору з урахуванням системних особливостей файлової системи. Однак, команда du має як переваги, так і недоліки. Тому в наступному розділі буде проведено порівняльний аналіз наявних інструментів для сканування файлової ієрархії APFS.

1.5 Порівняльний аналіз API для роботи з APFS (Apple File System)

У попередніх розділах ми розглянули інструменти для роботи з файловою системою Apple, зокрема NSFileManager, URLResourceKey та команду du. Кожен з цих інструментів має свої плюси і мінуси. У цій частині ми порівняємо ці інструменти, щоб визначити, який з них найкраще підходить для оптимізації сканування файлової системи APFS.

NSFileManager - це стандартний інструмент у Swift для керування файловою системою та отримання інформації про неї. Для роботи з одним файлом цей інтерфейс досить зручний і швидкий. Але для отримання розміру каталогів NSFileManager вимагає рекурсивного обходу всіх вкладених структур. Це значно уповільнює сканування файлової ієрархії. Також цей інтерфейс не надає інформації про фактичний обсяг дискового простору, що виділений для зберігання окремого файла. Це перешкоджає створенню універсального та ефективного рішення для сканування файлової системи APFS.

Команда du - це потужний інструмент для роботи з Unix/Linux системами. Дана команда має безліч параметрів, які можна використати під потреби користувача. Також за допомогою цього інструменту можна отримати фактичний розмір елементу файлової системи, а точніше кількість блоків, які він займає в системі. Команда du не вимагає реалізації рекурсивного підходу, що значно пришвидшує час сканування файлової системи. Проте, це термінальна команда, тому для її інтеграції в програмний код Swift потрібно використовувати такі класи, як Process та Pipe. Також, використовуючи цей інструмент, потрібно правильно обробити отримані дані та розбити їх на компоненти. Однак це можливо зробити лише після повного завершення виконання процесу командою du. Тому це може збільшити час сканування файлової системи. Додатково варто зазначити, що під час використання команди du можуть виникати помилки з доступом до певних каталогів або неправильними шляхами до елементів файлової системи. Отже, все це ставить певні виклики для розробки ефективного рішення для сканування Apple File System.

URLResourceKey - це безпечний та універсальний інструмент для взаємодії з файловою системою APFS. Він допомагає отримати інформацію про файли, директорії та томи файлової системи. Цей інструмент має велику різноманітність ключів, і це дає можливість отримати інформацію як про логічний, так і про фактичний розмір файлу. URLResourceKey також вимагає застосування рекурсивного обходу для сканування файлової ієрархії. Однак це відбувається набагато швидше в порівняні з NSFileManager, оскільки ми можемо обробляти лише необхідні нам атрибути і це пришвидшує запити до файлової системи. Крім того, на відміну від команди du, URL допомагає уникати помилок під час роботи з файловою системою. Процес обробки даних стає більш зручний та безпечний в порівнянні з командою du.

Отже, враховуючи все вище сказане, в контексті цього дослідження найбільш зручним інструментом для сканування Apple File System є URLResourceKey. Цей інтерфейс є більш ефективним, ніж NSFileManager або команда du. Він забезпечує швидку та надійну взаємодію з файловою системою, а за допомогою ключів fileAllocatedSizeKey та fileSizeKey можливо отримати усю необхідну інформацію.

Розділ 2. Методи сканування та багатопоточної обробки файлової ієрархії

2.1 Алгоритмічні стратегії обходу файлової системи

Для оптимізації сканування Apple File System ефективного та надійного API буде недостатньо, оскільки файлова ієрархія може бути дуже складна. Тому потрібно застосувати різні алгоритми обходу системи. Вони допоможуть врахувати особливості даної файлової системи та мінімізувати виконання зайвих операцій.

2.1.1 Верхньорівневий підхід

Найшвидший спосіб просканувати файлову систему - це застосувати верхньорівневий (Top Level) підхід. Даний алгоритм глибоко не занурюється в структуру директорії, а лише опрацьовує елементи, які знаходяться на верхньому рівні. Верхньорівневий (Top Level) підхід є досить швидким, бо мінімізує кількість звернень до файлової системи. Завдяки цьому алгоритму ми можемо швидко відсканувати систему, знайти на першому рівні файли та директорії, які займають найбільше дискового простору. Проте повної інформації про файлову ієрархію отримати неможливо.

Верхньорівневий підхід досить легко імплементується використовуючи команду du. Параметр -d (depth) дає можливість визначити необхідну глибину, на яку потрібно зануритись під час сканування файлової системи. Для даного підходу вказуємо глибину 1, і далі обробляємо отримані дані. Однак, під час використання URLResourceKey та NSFileManager для верхньорівневого підходу виникають труднощі, оскільки, як згадувалося вище, ці інструменти не мають окремих атрибутів для розміру директорії. Тому для реалізації верхньорівневого підходу, використовуючи URLResourceKey та NSFileManager, потрібно рекурсивного обходити директорії, які знаходяться на першому рівні. Отже, для цих інструментів верхньорівневий підхід не буде відрізнятись від повного

сканування файлової системи. Для URLResourceKey це не становить значної пробеми, бо цей інструмент дуже ефективною працює з Apple File System, і рекурсивний підхід майже не сповільнює сканування файлової системи. Проте для NSFileManager рекурсивний підхід значно зменшує швидкість сканування, особливо на складних файлових структурах, і це призводить до втрати ефективності верхньорівневого підходу.

2.1.2 Повний обхід файлової системи

Для того, щоб отримати повну інформацію щодо стану Apple File System та елементів цієї файлової системи потрібно застосувати повний обхід файлової системи (Full System Bypass). За допомогою даного підходу рекурсивно сканується файлова ієрархія та користувач отримує повний аналіз вмісту.

Використовуючи команду du, повний обхід файлової системи імплементується завдяки параметру -a (all). Ми отримуємо повний список усіх елементів файлової ієрархії та обробляємо його, розбиваючи на відповідні компоненти.

Повне сканування файлової системи за допомогою NSFileManager та URLResourceKey імплементується за допомогою ключів .size та fileAllocatedSizeKey відповідно. Рекурсивний алгоритм здійснює обхід кожного каталогу та його підкаталогів, і додає розміри усіх знайдених файлів.

Завдяки повному скануванню файлової системи ми можемо з високою точністю обчислити розмір директорій та проаналізувати кожен об’єкт. Проте час виконання буде досить значний, і користувач буде змушений довго очікувати результат. Тому більш зручним рішенням буде інтерактивний підхід, який допоможе зменшити час очікування для користувача. Даний підхід буде розглянуто у наступному розділі.

2.1.3 Інтерактивний обхід файлової системи

Під час застосування “лінивого” (lazy) обходу відбувається повне сканування файлової системи, але результати стають доступними для користувача поступово, в міру їх опрацювання. Тому користувач зможе отримати перші результати перш, ніж процес буде повністю завершений. Даний підхід значно покращує взаємодію з користувачем та підвищує продуктивність сканування.

Для реалізації “лінивого” обходу було використано AsyncStream. Дана структура забезпечує зручний спосіб створення асинхронної послідовності без ручної реалізації асинхронного ітератора. AsyncStream створюється за допомогою замикання (closure). У межах замикання файли або директорії додаються в потік, використовуючи метод yield(_:). Це дозволяє передавати об’єкти без необхідності очікувати завершення сканування всієї файлової системи. Коли всі елементи передані у потік викликається метод finish() для завершення потоку. Інтерактивний обхід забезпечує гнучкість, і елементи можуть бути опрацьовані вибірково. Однак якщо обробка даних займає багато часу, потік може блокуватися.

Отже, інтерактивний обхід дозволяє зробити сканування файлової системи більш динамічним та зручним для користувача.

2.2 Фільтрація за стоп-словами

Крім вищезгаданих підходів, для оптимізація файлової системи було імплементовано підхід фільтрації за стоп-словами. Фільтрація дозволяє виключити певні директорії або файли, які не важливі для результату. Це можуть бути системні каталоги, тимчасові файли тощо. Фільтрація за стоп-словами реалізована двома способами: з використанням алгоритму Depth-First Search (DFS) та з використанням алгоритму Breadth-First Search (BFS). Обидві

реалізації імплементовані за допомогою протоколу TraversalAlgorithm. Далі розглянемо детальніше кожен з цих способів.

2.2.1 Фільтрація за стоп-словами з використанням алгоритму Depth-First Search (DFS)

В першій реалізації фільтрації за стоп-словами було використано алгоритм Depth-First Search (DFS). Алгоритм DFS - це пошуковий алгоритм, в якому пошук відбувається шляхом розширення першого дочірнього вузла кореня, який з'являється, і таким чином заглиблюється все глибше і глибше, поки не буде знайдено потрібний вузол, або поки він не досягне вузла, який не має дочірніх вузлів. Тоді пошук повертається назад, повертаючись до останньої вершини, яку ще не було досліджено. У нерекурсивній реалізації всі щойно відкриті вершини додаються до стеку для дослідження [10].

Для фільтрації за стоп-словами з використанням DFS в даному досліджені реалізовано клас DFSAlgorithm, який відповідає протоколу TraversalAlgorithm. Алгоритм рекурсивно проходить усі директорії файлової структури та перевіряє наявність стоп-слова у назві директорії або файлу. Якщо стоп-слово було знайдено, алгоритм ігнорує цей елемент і сканування продовжується. Якщо елемент не містить стоп-слова, він обробляється за допомогою API, які було згадано вище.

За допомогою даного підходу ми можемо оптимізувати сканування файлової системи APFS, ефективно обробити складні та глибокі файлові структури. Також алгоритм не буде акцентувати увагу на непотрібних елементах, і це дозволить зменшити час сканування.

2.2.2 Фільтрація за стоп-словами з використанням алгоритму Breadth-First Search (BFS)

В наступній реалізації фільтрації за стоп-словами було використано алгоритм Breadth-First Search (BFS). Алгоритм BFS - це алгоритм обходу, який

зазвичай використовується для пошуку або обходу графів або деревовидних структур даних. Ідея полягає в тому, щоб дослідити всі вузли на поточному рівні глибини, перш ніж переходити на наступний рівень. Такий обхід досягається шляхом відвідування всіх сусідів вузла перед тим, як перейти до сусідніх вершин наступного рівня [11].

Для фільтрації за стоп-словами з використанням BFS реалізовано клас BFSAlgorithm, який, як і клас DFSAlgorithm, відповідає протоколу TraversalAlgorithm. Алгоритм BFS дістає елемент дістає елемент з черги на перевіряє його на наявність стоп-слів. Якщо назва поточного файлу або директорії не містить стоп-слів, елемент файлової системи обробляється, а його піддиректорії додаються до черги для подальшої обробки. Якщо поточний елемент не містить стоп-слів, алгоритм ігнорує його.

На практиці для сканування файлової системи DFS алгоритм працює швидше, ніж BFS. Проте BFS все ж є досить ефективним алгоритмом та дозволяє швидше проаналізувати елементи файлової системи на верхнім рівнях.

2.3 Підходи до багатопоточної обробки файлової ієрархії

Послідовна обробка не завжди є ефективною, особливо якщо структура даних досить складна та багаторівнева. Тому для оптимізації та прискоренню сканування файлової системи APFS може бути використано багатопоточна обробка файлової ієрархії. Swift включає підтримку паралелізму, що дозволяє писати код, який може обробляти декілька функцій нібито одночасно, що покращує швидкість реагування та ефективність додатків. Класичними методами для реалізації багатопоточної обробки є Swift Concurrency та Grand Central Dispatch (GCD). Для реалізації послідовної та багатопоточної обробки файлових структур було імплементовано протокол ConcurrencyApproach. Далі детально буде розглянуто кожен з методів обробки.

2.3.1 Послідовна обробка файлової ієрархії

Найпростішим варіантом обробки файлової ієрархії є послідовна обробка. Вона означає обробку всіх операцій в одному потоці. Кожен елемент файлової системи опрацьовується поступово, тому даний спосіб не потребує механізму управління потоками та синхронізації. Для реалізації послідовного методу обробки було створено клас SingleProcessing, який відповідає протоколу ConcurrencyApproach.

Послідовна обробка дуже проста в реалізації, а також є можливість легко відлагоджувати програмне забезпечення. Для окремих файлів або невеликих директорій даний метод є досить зручним. Однак, для більшості реальних файлових ієрархій цей метод поступається багатопоточній обробці.

2.3.2 Паралельна обробка за допомогою Swift Concurrency

Swift Concurrency - це вбудована підтримка написання асинхронного та паралельного коду у структурований спосіб. Даний інструмент був впроваджений починаючи з версії Swift 5.5. Swift Concurrency використовує сучасні можливості мови, такі як async/await, структурований паралелізм та Swift Actors для ефективного керування паралельними завданнями, гарантуючи, що додаток залишатиметься швидким та продуктивним.

Асинхронний код дозволяє додатку виконувати завдання, не блокуючи головний потік. У Swift асинхронні функції визначаються за допомогою ключового слова async. Ці функції можуть призупиняти виконання в точках призупинення і відновлювати його пізніше. Ключове слово await використовується для виклику асинхронних функцій, сигналізуючи, що функція буде чекати на завершення операції, перш ніж продовжити.

Tasks у Swift Concurrency - це блоки роботи, які виконуються асинхронно. Вони надають можливість виконувати асинхронні операції, не блокуючи

поточний потік. Task Groups дозволяють створювати та керувати кількома завданнями, які працюють разом для отримання загального результату. Така організація допомагає ефективніше керувати паралелізмом, забезпечуючи структурований спосіб одночасного виконання та керування кількома асинхронними операціями.

Для реалізації сканування Apple File System було використано функцію withThrowingTaskGroup. Дана функція в Swift Concurrency спеціально розроблена для обробки одночасного виконання декількох задач, які можуть спричиняти помилки. Основна мета withThrowingTaskGroup - дозволити паралельно виконання кількох дочірніх задач у Task Group, а також надати механізм для обробки помилок, які може спричинити будь-яке з дочірніх завдань. Якщо дочірня задача спричиняє помилку, функція withThrowingTaskGroup перехоплює цю помилку і дозволяє поширити її за межі тіла TaskGroup. Це означає, що ми можемо обробляти помилки з декількох задач в одному місці, замість того, щоб обробляти їх окремо в кожній задачі.

Паралельна обробка робить сканування ефективнішим, але можуть виникати такі проблеми, як data race. Ця проблема виникає, коли декілька потоків намагаються одночасно модифікувати або отримати доступ до спільного ресурсу, що призводить до непередбачуваних результатів. Найбільш класичний спосіб уникнути data race - це використання механізму NSLock. Він гарантує, що тільки один потік може змінювати елементи одночасно. NSLock - це простий та ефективний метод захист, але при необережному використанні він може призвести до взаємного блокування (deadlock). Взаємне блокування виникає, коли два або більше потоків очікують один одного, щоб зняти блокування, що призводить до зупинки програми на невизначений час. Тому Swift Concurrency надає більш зручний та ефективний механізм Actors, який гарантує, що всі операції над елементами виконуються послідовно і безпечно. Даний механізм дозволяє уникнути багатьох складнощів, пов'язаних з ручною синхронізацією потоків. Використовуючи Actors, не потрібно вручну блокувати та

розблоковувати ресурси для забезпечення безпеки потоків, вони автоматично керують доступом до свого внутрішнього стану.

2.3.3 Паралельна обробка за допомогою Grand Central Dispatch (GCD)

Grand Central Dispatch (GCD) - це низькорівневий API для керування паралельними задачами в додатках для iOS та macOS. GCD керує чергами виконання (DispatchQueue), які відповідають за паралельне або послідовне виконання завдань, розподіляючи системні ресурси і потоки оптимальним чином на основі доступних системних ресурсів і поточного завантаження системи. Також існує декілька глобальних черг (DispatchQueue.global), кожна з яких має різний рівень якості обслуговування (QoS), який використовується для визначення пріоритетності виконання завдань у черзі. Також існує такий механізм, як групи задач (DispatchGroup), які дозволяють об'єднати набір задач і синхронізувати поведінку в групі. Цей механізм надає можливість приєднати кілька робочих елементів до групи і запланувати їх асинхронне виконання в одній черзі або в різних чергах.

У даному дослідженні для сканування файлової системи APFS використовується як Grand Central Dispatch, так і Swift Concurrency. Для поєднання та підтримки обох методів паралельної обробки використовується функція withCheckedThrowingContinuation. Вона допомагає легко інтегрувати старі API з сучасним async/await підходом, а також виявляти потенційні помилки на ранніх стадіях за допомогою вбудованих перевірок Swift.

Незважаючи на новий підхід обробки, який використовує Swift Concurrency, GCD залишається дуже ефективним інструментом завдяки точному контролю над виконанням задач та підтримці застарілих систем.

2.4 Оцінка ефективності різних підходів

У цьому розділі ми порівняємо ефективність методів, які були розглянуті вище. Основними критеріями для оцінки ефективності різних підходів для сканування файлової системи APFS є швидкість, точність, масштабованість та використання ресурсів.

Швидкість – це час який необхідний для всіх елементів файлової ієрархії та є основний показник ефективності будь-якого підходу для сканування файлової системи. Послідовний підхід є найповільнішим, бо всі задачі виконується в одному потоці. Паралельна обробка значно прискорює обробку за рахунок розподілення задач між ядрами процесора. Swift Concurrency працює дещо швидше за GCD через оптимізацію для асинхронних операцій. Серед алгоритмічних стратегій найшвидшим є верхньорівневий підхід, бо він обробляє лише елементи верхнього рівня. Повний підхід та інтерактивний підхід працюють значно повільніше, оскільки вони рекурсивно занурюються у директорії файлової системи. Проте, з точки зору користувача, інтерактивний підхід є швидшим, оскільки він надає результати поступово, і користувач не повинен чекати повного завершення процесу обробки.

Наступним критерієм є точність, яка визначає наскільки коректно виконується сканування файлової системи. Послідовна та паралельна обробка забезпечують високу точність, але паралельна обробка вимагає додаткових механізмів для синхронізації доступу до спільних ресурсів. Серед алгоритмічних стратегій найбільшим точними є повний та інтерактивний обхід, оскільки вони сканують усі рівні файлової ієрархії. Верхньорівневий підхід є менш точним та не відображає повної інформації про систему.

Масштабованість - здатність методу ефективно обробляти великі ієрархії файлів. Послідовна обробка має низьку масштабованість. Паралельна обробка розподіляє задачі між процесами, через що має досить високу масштабованість. Також варто зауважити, що Swift Concurrency працює краще через оптимізацію асинхронних операцій. Серед алгоритмічних стратегій повний та інтерактивний

обхід мають приблизно однакову масштабованість, але інтерактивний прохід показує результати сканування частинами в міру завершення їх обробки.

Останнім критерієм для оцінки є використання ресурсів. Даний критерій показує наскільки ефективно поточний метод використовує ресурси системи: пам’ять, процесор, диск. Послідовна обробка не є ефективною для сучасних багатоядрених систем. Обидва паралельних підходи ефективно використовують всі доступні ядра процесора, але GCD потребує більш ретельного управління ресурсами через необхідність синхронізації. Серед алгоритмічних стратегій мінімальну кількість ресурсів використовує верхньорівневий підхід. Повний та інтерактивний обхід використовують значно більше ресурсів, оскільки вони рекурсивно сканують файлову систему.

Отже, оцінка ефективності підходів, які були застосовані в даному досліджені показала, що кожен метод має свої переваги та недоліки. Тому кожен підхід може бути продуктивним для різних типів задач.

Розділ 3. Архітектура та порівняльний аналіз розробленого рішення

3.1 Архітектура розробленого рішення

Рішення, яке було розроблене в ході даного дослідження, базується на модульному підході, що забезпечує гнучкість та масштабованість. Нижче можна побачити діаграму, яка відображає архітектуру створеного рішення (Рис 3.1).

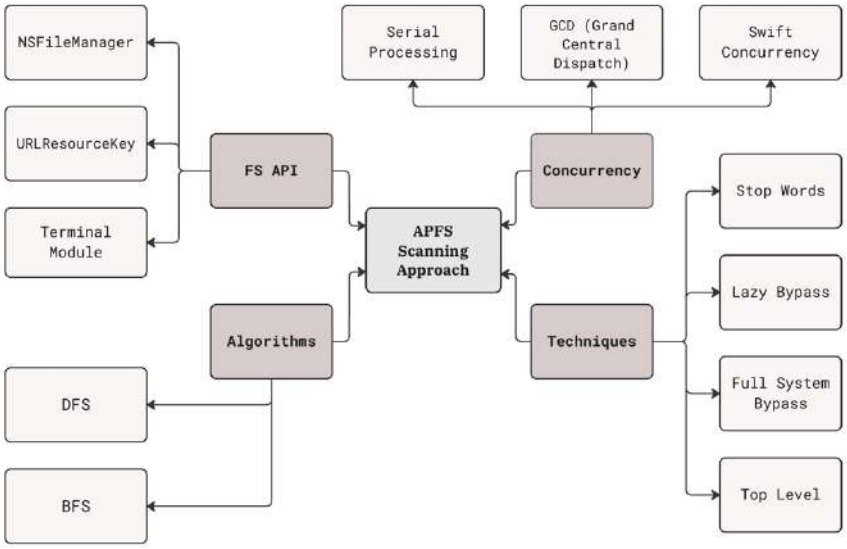

Рис 3.1 Структура розробленого рішення

Центральним елементом даної архітектури є протокол FSAPI, який визначає стандартний набір методів для взаємодії з файловою системою APFS. Реалізують цей протокол три класи, які використовують різні API для роботи з файловою системою:

- FileManagerAPI - клас, який використовує NSFileManager
- ResourceValuesAPI - клас, який використовує URLResourceKey
- DuCommandAPI - клас, який використовує термінальну команду du

Даний підхід дає можливість легко комбінувати різні методи взаємодії з файловою системою в залежності від потреб.

Для підвищення продуктивності сканування файлової структури було створено протокол ConcurrencyApproach, який також реалізують три класи:

- SingleProcessing - клас для послідовної обробки файлової ієрархії
- SwiftConcurrencyApproach - клас для паралельної обробки за допомогою Swift Concurrency
- GCDApproach - клас для паралельної обробки за допомогою Grand Central Dispatch

Для аналізу файлової ієрархії імплементовано різні стратегії обходу, які визначені за допомогою протоколу TraversalTechnique:

- TopLevelTechnique - клас, який реалізує верхньорівневий підхід
- FullSystemBypassTechnique - клас, який реалізує повний обхід файлової системи
- StopWordsTechnique - клас, який реалізує фільтрацію за стоп-словами

Також для фільтрації за стоп-словами використовуються пошукові алгоритми, реалізовані в класах BFSAlgorithm та DFSAlgorithm, які відповідають протоколу TraversalAlgorithm.

Основний клас для сканування файлової системи - це SizeCalculator. Він виконує звичайне сканування за допомогою функції calculateSize, використовуючи обрану стратегію обходу, підхід багатопоточної обробки та API для взаємодії з системою. Також для реалізації інтерактивного підходу, який був описаний у попередніх розділах, створений окремий метод calculateSizeLazily.

Система є модульною, масштабованою та ефективною, тому ми можемо легко змінювати алгоритми та підходи для сканування Apple File System.

3.2 Оптимізації сканування файлової системи

У попередньому розділі було розглянуто архітектуру розробленого рішення. Далі ми розглянемо основні оптимізації, які були використані в даному дослідженні.

В цій роботі імплементовано три незалежні підходи для взаємодії з файлової системою APFS. Кожен з них має певні плюс та мінуси, тому користувач має можливість обрати необхідний підхід для конкретної ситуації.

Один з ключових недоліків традиційного сканування – необхідність очікувати повного завершення процесу для отримання результатів. Для усунення цього недоліку було реалізовано інтерактивний підхід, який поступово передає данні. Завдяки цьому користувач може одразу ознайомитись з певними результатами після їх обробки, і це, безумовно, покращує користувацький досвід.

Ще однією оптимізацією в розробленому рішенні є паралельна обробка елементів файлової системи. Це значно пришвидшує процес сканування і робить його набагато ефективнішим. Імплементовано три підходи: звичайна послідовна обробка та дві паралельні обробки за допомогою Swift Concurrency та Grand Central Dispatch. Розроблене рішення надає користувачу можливість обрати необхідний для нього варіант.

Для того, щоб зменшити навантаження на файлову систему та уникнути обробки непотрібних елементів було створено фільтрацію за стоп-словами. Вона використовує алгоритми BFS та DFS для сканування системи, та не обробляє непотрібні файли та директорії. За допомогою фільтрації ми можемо зменшити час обробки файлової ієрархії та зосередитись лише на необхідних даних.

Окрім фільтрації за стоп-словами та “лінивого” підходу, було імплементовано й інші методи такі як верхньорівневий підхід та повне сканування файлової системи. Це дає гнучкість під час сканування файлової системи та дозволяє вибрати оптимальний метод в залежності від поставленої задачі. Верхньорівневий підхід забезпечує максимально швидкий огляд файлової ієрархії, а повне сканування надає максимально точні результати, але потребує більше часу. Тому розроблене рішення дозволяє досягти балансу між швидкістю та глибиною аналізу.

3.3 Оцінка та аналіз проведених тестувань

Для оцінки ефективності методів і алгоритмів сканування Apple File System, описаних у попередніх розділах, було проведено комплексне тестування з використанням спеціально створених тестових наборів даних. Вони моделюють різні структури файлових ієрархій, і це дозволяє оцінити продуктивність алгоритмів у реальних сценаріях.

Тестування проводилося з використанням наборів даних, які створювались за допомогою генератора даних. Цей інструмент дозволяє створювати файлові ієрархії різних шаблонів, які відображають типові та крайні випадки файлових структур у APFS. Генератор даних включає чотири основні шаблони: Breadth-First Structure (BFS), Depth-First Structure (DFS), Balanced Tree Structure та Unbalanced Tree Structure. Кожен шаблон має унікальні характеристики, які допомагають протестувати різні аспекти алгоритмів сканування:

- Breadth-First Structure (BFS) – шаблон, який горизонтально розширюється перед поглибленням. Він допомагає оцінити продуктивність сканування широких ієрархій з великою кількістю дочірніх елементів.
- Depth-First Structure (DFS) – шаблон, який допомагає перевірити роботу алгоритмів під час сканування глибоко вкладених каталогів.
- Balanced Tree Structure – шаблон, який моделює передбачувану структуру та має рівномірно розподілену ієрархію файлів.
- Unbalanced Tree Structure – шаблон, який допомагає оцінити сканування нерегулярних ієрархій, що імітують реальні сценарії.

Для тестування використовувався комп`ютер з macOS Sequoia 15.4.1, а також він оснащений SSD накопичувачем, який працює з APFS. Кожен алгоритм та підхід сканування файлової системи тестувався на всіх чотирьох шаблонах даних. У згенерованих файлових ієрархіях розмір файлів був різний: від порожніх файлів до файлів з випадковим розміром. Кожен алгоритм оцінювався

за часом виконання та піковим використанням пам’яті під час сканування кожного набору даних.

Для наочної ілюстрації результатів тестування було побудовано декілька графіків тих чи інших алгоритмів та підходів для сканування Apple File System. Для початку було проведено порівняння API для роботи з APFS за часом виконання сканування чотирьох типів файлової ієрархії: BFS, DFS, Balanced та Unbalance (Рис 3.2).

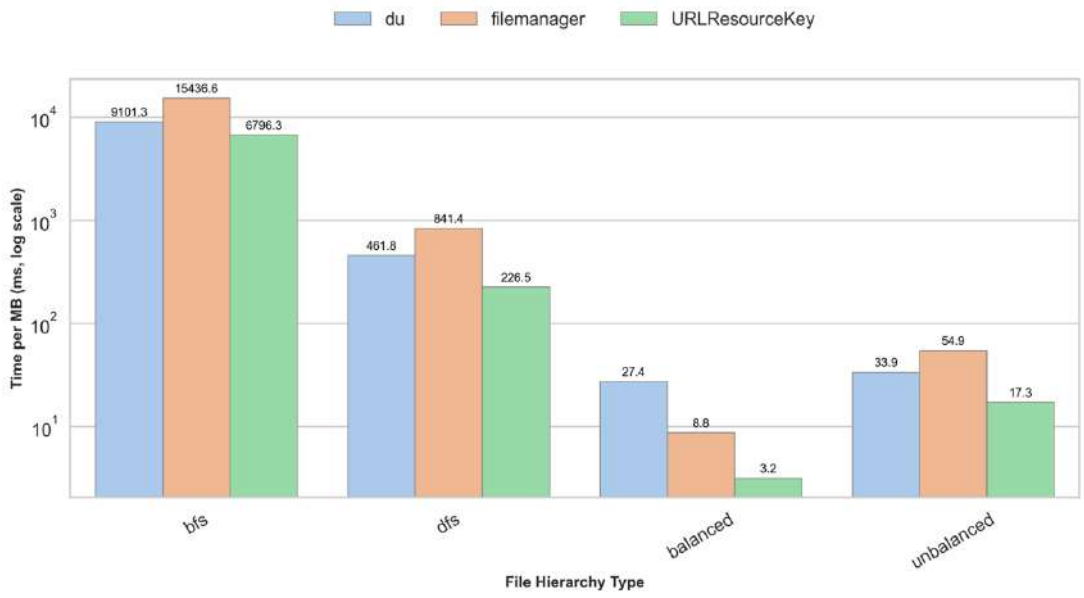

Рис 3.2 Порівняння часу виконання сканування за допомогою різних API для чотирьох типів файлових ієрархій

Графік демонструє значні відмінності в продуктивності методів сканування залежно від структури файлової ієрархії. Найскладнішим для обробки виявився тип Breadth-First Structure. Широка структура цього типу файлової ієрархії з великою кількістю дочірніх елементів значно ускладнила сканування та вимагала більше часу. Натомість найлегшим типом є збалансована ієрархія Balanced Tree Structure, завдяки її передбачуваній і рівномірній організації. Найшвидшим API для всіх чотирьох типів структур є URLResourceKey, який в кілька разів перевершував NSFileManager та команду du для всіх чотирьох типів файлових структур. Другою за ефективністю є

команда du, яка показала стабільні результати, але поступилась NSFileManager лише під час сканування збалансованої структури Balanced Tree.

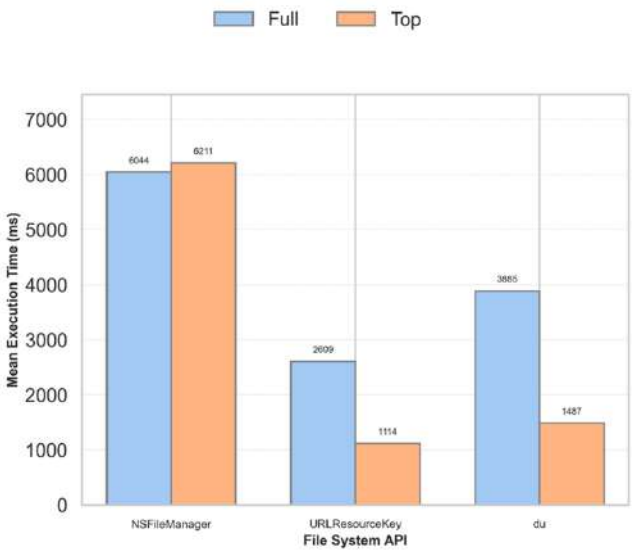

Рис 3.3 Порівняння часу виконання сканування за допомогою повного та вернхньорівневого обходу та різних API

На Рис 3.3 наведено діаграму порівняння середнього часу сканування за допомогою повного обходу та верхньорівневого підходу з використанням NSFileManager, URLResourceKey та команди du. Верхньорівневий підхід підвищує продуктивність та скорочує час сканування у 2-3 рази для URLResourceKey та команди du. Однак для NSFileManager результати залишаються незмінними, бо даний інструмент вимагає використання рекурсивного обходу для обох алгоритмів обходу файлової ієрархії.

Інтерактивний обхід і фільтрація за стоп-словами також продемонстрували відмінні результати. Інтерактивний обхід починає видавати перші результати швидше за інші алгоритми, при цьому не втрачаючи точності. Фільтрація за стоп-словами дозволяє обробляти лише необхідні елементи файлової ієрархії, в той же час скорочує час сканування та споживання системних ресурсів.

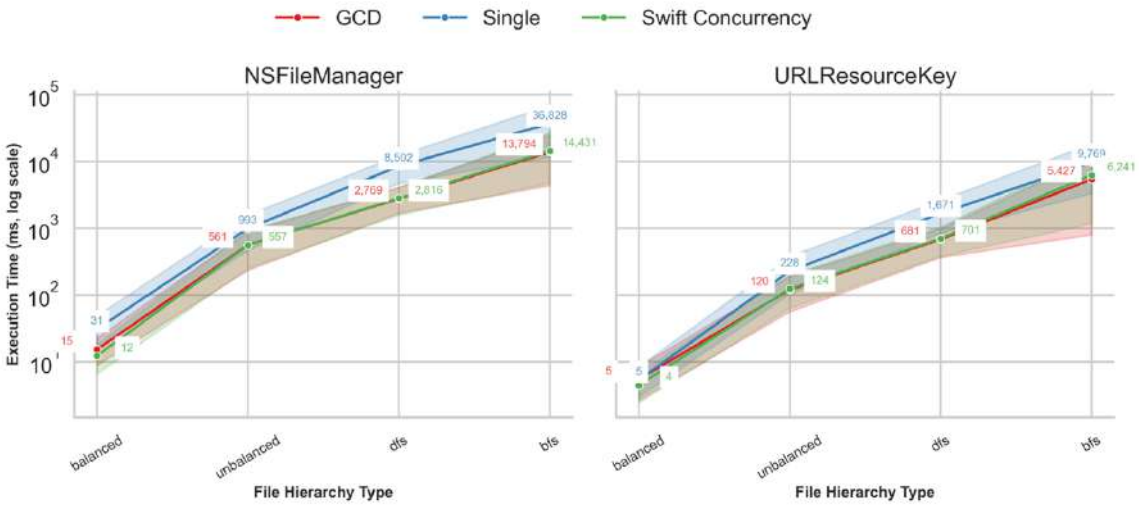

Рис 3.4 Порівняння середнього часу виконання для послідовної обробки, GCD та Swift Concurrency

Рис 3.4 відображає графік порівняння підходів до багатопоточної обробки для усіх чотирьох типів файлових структур. Графік розподілено на дві частини: ліва відображає результати для NSFileManager, а права – для URLResourceKey. Як видно, багатопоточні методи обробки CGD та Swift Concurrency набагато ефективніші, ніж однопоточна обробка. Вони прискорюють сканування APFS у 2-3 рази. Крім того, GCD і Swift Concurrency демонструють практично ідентичний рівень продуктивності.

File System API	**GCD (MB)**	**Single (MB)**	**Swift concurrency (MB)**
NSFileManager	222.0	226.5	241.1
URLResourceKey	96.4	70.2	113.9
du	59.9	43.5	65.7

Таблиця 3.1 Середнє використання оперативної пам'яті різними багатопоточними підходами та API

Ще одним параметром під час аналізу ефективності розробленого рішення було використання оперативної пам’яті методами сканування. Таблиця 3.1 демонструє використання пам’яті під час сканування APFS за допомогою NSFileManager, URLResourceKey, команди du та різних методів багатопоточної обробки. NSFileManager споживає найбільше пам’яті, в той же час команда du є найбільш ефективною та демонструє найнижчий рівень використання. Крім того, методи багатопоточної обробки також мають значний вплив на використання пам’яті. Swift concurrency демонструє найбільше використання пам’яті, а Grand Central Dispatch дещо менші. Найменше навантаження створює послідовна обробка. Хоча вона й вимагає найменше ресурсів, але методи багатопоточної обробки значно підвищують швидкість сканування, а додаткове використання пам’яті компенсується значним збільшенням продуктивності.

3.4 Можливості для вдосконалення

Рішення, яке було розроблене в ході даного дослідження, значно оптимізує сканування файлової системи APFS та покращує продуктивність обробки даних. Однак завжди існують можливості для подальшого вдосконалення.

Одним з потенційних напрямків для покращення поточного рішення – це використання прямої взаємодії з файловою системою APFS за допомогою Apple File System Reference. Це низькорівневий API, який надає доступ до внутрішніх механізмів файлової системи APFS. Проте існує дуже багато тонкощів пов’язаних в структурі Apple File System, що безумовно ускладнює реалізацію даного методу. Більше того, документація Apple File System Reference обмежена і тому процес розробки стає набагато складнішим.

Розроблене рішення підтримує сканування лише файлової системи APFS. Однак модульна архітектура дозволяє розширити поточне рішення для підтримки інших файлових систем. Це б дозволило використовувати його в різних середовищах.

ВИСНОВКИ

Для дослідження ефективності різних методів оптимізації було розроблено застосунок для сканування файлової системи Apple File System. Його модульна архітектура забезпечує гнучкість, масштабованість та можливість комбінувати різні підходи. Застосунок дозволив провести детальний аналіз, дослідити різні методи сканування, порівняти їх ефективність і визначити стратегії, що найкраще підходять для різних сценаріїв використання.

Однією з частин реалізації став протокол FSAPI, який дозволяє використовувати сучасні API та легко змінювати методи взаємодії з файловою системою, такі як NSFileManager, URLResourceKey або команда du. У даному рішенні імплементовано різні стратегії обходу файлової системи, такі як верхньорівневий підхід, повний обхід, інтерактивний обхід та фільтрація за стоп-словами. Варіативність цих підходів надає можливість гнучко налаштовувати процес сканування та підвищує його ефективність. Також було розроблено послідовну обробку елементів файлової системи та два методи для багатопоточної обробки за допомогою сучасних засобів мови Swift, таких як Swift Concurrency та Grand Central Dispatch.

Розроблений інструмент був протестований на 4 варіантах штучно створених файлових ієрархій, і це дозволило оцінити його продуктивність та точність. Результати тестування показали ефективність застосованих підходів до сканування, оптимізації та багатопоточної обробки. Запропоновані методи дозволили зменшити середній час обходу Apple File System та скоротити кількість непотрібних запитів до файлової системи. Зокрема, використання паралельної обробки за допомогою GCD та Swift Concurrency дозволило зменшити час сканування в 2–3 рази. Для більшості типових файлових структур, характерних для користувацьких систем, найкращі результати продемонструвала комбінація Swift Concurrency та URLResourceKey.

В ході дослідження було знайдено декілька способів покращення розробленого рішення, що свідчить про його потенціал для подальшого

розвитку. У перспективі запропоновані методи можуть знайти застосування не лише для оптимізації сканування файлової системи APFS, а й для більш глибокого розуміння процесів управління даними в сучасних файлових системах macOS.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Tamura E., Giampaolo D. Introducing Apple File System. Apple Inc. 2016.
2. Apple Inc. HFS Plus Volume Format: Technical Note TN1150. 2010.
3. Rane R., Singh, A. Demystifying File Systems: A Comprehensive Exploration of Data Organization. 2024.
4. Apple Inc. Apple File System Guide. 04.06.2018. URL: https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/APFS_Guide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40016999-CH1-DontLinkElementID_15.
5. Hansen K. H., Toolan F. Decoding the APFS file system. *Digital Investigation*. 2017. Vol. 22. P. 107–132.
6. Apple Inc. FileManager. Apple Developer. URL: https://developer.apple.com/documentation/foundation/filemanager.
7. Apple Inc. URLResourceKey. Apple Developer. URL: https://developer.apple.com/documentation/foundation/urlresourcekey.
8. Platt D. Tweak Your Mac Terminal. Berkeley, CA : Apress, 2021.
9. Apple Inc. Streams, Sockets, and Ports. Apple Developer. URL: https://developer.apple.com/documentation/foundation/streams_sockets_and_ports.
10. Garg D., Kaur N. Analysis of the Depth First Search Algorithms. *Computer Science and Engineering Department, Thapar University.* 2012.
11. Holdsworth H. The Nature of Breadth-First Search. *School of Computer Science Mathematics and Physics, James Cook University, Australia*. 1999.
12. Apple Inc. Apple File System Reference. Apple Developer. URL: https://developer.apple.com/support/downloads/Apple-File-System-Reference.pdf.