DOI: https://doi.org/10.36910/6775-2524-0560-2025-60-13
УДК: 004.42:004.415:004.75
Даниленко Станіслав Дмитрович, аспірант
https://orcid.org/0000-0002-8142-3018
Смеляков Сергій Вячеславович, д.ф.-м.н, професор
https://orcid.org/0000-0002-5791-2479
Харківський національний університет радіоелектроніки, м. Харків, Україна

ФОРМАЛІЗОВАНИЙ МЕТОД ОПТИМІЗАЦІЇ ВИКОРИСТАННЯ ОПЕРАТИВНОЇ ПАМ'ЯТІ В СИСТЕМАХ ПОШУКУ ЗОБРАЖЕНЬ НА ОСНОВІ ВМІСТУ

Даниленко С. Д., Смеляков С. В. Формалізований метод оптимізації використання оперативної пам'яті в системах пошуку зображень на основі вмісту. У статті запропоновано формалізований метод оптимізації використання оперативної пам'яті в системах пошуку зображень за вмістом, реалізованих мовою Java. Метод поєднує два підходи: низькорівневу оптимізацію структур даних та компіляцію застосунку у нативний код за допомогою GraalVM. В основі методу – аналіз типових структур, які використовуються для зберігання дескрипторів зображень, і їх заміна на ефективні по використанню пам'яті еквіваленти на рівні мови програмування. Крім того, нативна компіляція дозволяє скоротити додаткові витрати на виконання і зменшити загальне споживання пам'яті під час виконання програмного коду. Метод оцінено на прикладі моделі Multidimensional Cube, що передбачає зберігання векторних дескрипторів у оперативній пам'яті. Наведено теоретичні формули для оцінки використання пам'яті залежно від структури представлення даних, а також результати експериментального профілювання. Вказано на недоліки та обмеження, які виникають при використанні запропонованого методу: збільшення часу та кількості ресурсів на етапі компіляції та неможливість використання деяких програмних бібліотек. Оптимізована реалізація демонструє до 3-кратного зниження використання оперативної пам'яті і прискорення обробки, а GraalVM забезпечує додаткове зменшення до 73%. Запропонований метод може бути застосований в інших CBIR-системах і в ширшому класі задач, пов'язаних із зберіганням великої кількості векторних структур у пам'яті. Оскільки інші мови програмування мають подібні структури даних та/або підтримку зі сторони GraalVM.
Ключові слова: інформаційні технології, оптимізація продуктивності, типи даних, структури даних, оперативна пам'ять, пошук зображень на основі вмісту, GraalVM, Java, нативна компіляція

Danylenko S., Smelyakov S. Formalized method for RAM optimization in content-based image retrieval systems. This paper presents a formalized method for optimizing RAM usage in content-based image retrieval systems implemented in Java. The method combines two approaches: low-level optimization of data structures and compilation of the application into native code using GraalVM. At its core, the method analyzes commonly used structures for storing image descriptors and replaces them with memory-efficient equivalents at the programming language level. Additionally, native compilation reduces runtime overhead and overall memory consumption during program execution. The method is evaluated using the Multidimensional Cube model, which stores vector descriptors entirely in memory. The paper provides theoretical formulas for estimating memory usage based on data representation structures and presents results of experimental profiling. It also discusses the limitations of the proposed method, including increased compilation time and resource consumption, as well as incompatibility with some software libraries. The optimized implementation shows up to a threefold reduction in RAM usage and improved processing speed, while GraalVM provides an additional runtime memory reduction of up to 73%. The method can be applied to other CBIR systems and broader classes of problems that involve storing large volumes of vector structures in memory, especially when using programming languages with similar data structures and/or GraalVM support.
Keywords: information technology, performance optimization, data types, data structures, RAM, CBIR, Java, GraalVM, native compilation.

Вступ. Пошук інформації в мережі Інтернет виконується на основі запитів, які створюють користувачі пошукових систем. Пошуковий запит не завжди має текстове представлення. Наприклад, людина може голосом казати те, що їй потрібно знайти, і спеціальні моделі будуть виконувати пошук на основі цієї інформації [1]. Також пошук може відбуватися на основі графічної інформації. Користувач завантажує зображення чи робить фото, на основі якого потрібно знайти подібні зображення. Такий пошук називається візуальним пошуком чи пошуком зображень на основі вмісту (CBIR) [2].

Потреба в такому пошуку є досить широкою, адже його можна використовувати в різних сферах, наприклад: в пошуку товарів на основі зображень, електронній комерції [3], медичній діагностиці та встановленні діагнозів [4], ідентифікації людей тощо [2].

Статистика використання функції CBIR в популярних пошукових системах підтверджує це. Наприклад, у пошуковій системі Google щоденно виконується 13.7 мільярдів запитів. Серед яких 10.62% результатів пошуку є зображеннями. Функцію пошуку на основі вмісту зображення використовують приблизно 12 мільярдів разів у місяць [5].

Пошук зображень виконується на основі спеціального представлення зображень у вигляді дескрипторів. Зазвичай це числові вектори певної довжини, які описують певні характеристики

зображення [6]. Під час пошуку відбувається їх порівняння. Дескриптори займають значно менше місця ніж відповідні їм зображення, але все одно багато. Це пов'язано з сучасними об'ємами даних. Велику кількість систем можна віднести до категорії Big Data, де мова може йти про петабайти інформації, чи навіть більше [7].

Тому актуальною є проблема економії ресурсів, які використовуються високонавантаженими системами. Такі системи часто застосовується хмарні обчислення, вартість використання яких є досить високою [8, 9]. В контексті CBIR-систем, необхідна кількість ресурсів, що потребує пошукова система, безпосередньо залежить від ефективності реалізації дескрипторів, які використовуються, та від ефективності реалізації пошукової моделі, як на концептуальному рівні, так і на рівні реалізації програмного коду.

Аналіз останніх досліджень і публікацій. В основі пошукової системи зображень знаходиться пошукова модель, яка оперує дескрипторами: аналізує зображення, створює дескриптори, розміщує їх певним чином, обробляє пошуковий запит, порівнює дескриптори зображень та формує результат. На концептуальному рівні реалізації CBIR-моделей досліджується 2 основні напрями: 1) зберігання векторів дескрипторів під час всього періоду роботи системи і їх використання для пошуку; 2) перетворення вхідних векторів дескрипторів до спрощеного вигляду і використання процедури наближеного відтворення значення для використання під час пошуку. Прикладами реалізації цих підходів є Asymmetric Distance Computation та Symmetric Distance Computation з моделі Product Quantization відповідно [10]. Зберігання початкових векторів є більш пріоритетним та рекомендованим до використання, адже забезпечує вищу точність пошуку. Однак кількість пам'яті, необхідною для зберігання дескрипторів в цьому випадку також є більшою.

Однією з основних задач CBIR є зменшення пошукового простору. Реалізація моделей може базуватися на різних підходах та структурах даних, таких як: графи, дерева, хеш, кластеризація тощо [2]. Наприклад, модель Multidimensional Cube (MDC) поділяє простір властивостей дескриптора на підпростори і представляє дані у вигляді багатовимірного кубу. Найефективнішим в плані процесу пошуку є підхід, коли дескриптори зберігаються в оперативній пам'яті (RAM) під час роботи системи [11]. Тому ключовим є питання мінімізації необхідної кількості RAM.

Різні CBIR-задачі потребують спеціалізовані дескриптори, які можуть зберігати інформацію про різні деталі зображення і різну кількість інформації [6]. Дана робота не досліджує ефективність різних дескрипторів, а концентрується на дослідженні використання існуючих дескрипторів та оптимізації ефективності реалізації моделі CBIR. А саме рішень, прийнятих при написанні програмного коду, на прикладі моделі MDC. Модель реалізована мовою програмування Java, тому розглядаються і аналізуються оптимізації для неї.

Однією з ключових задач в досягненні оптимізації використання RAM у програмній реалізації є ліквідація втрат пам'яті. Вони виникають, коли об'єкти живуть в пам'яті довше, ніж дійсно потрібні. Це може бути пов'язано з використанням: некерованих посилань на об'єкти, перевантажених статичних полів, некоректну конфігурацію чи вибір збирача сміття (GC) [12, 13].

Активно досліджується ефективність роботи програмних рішень з використання різних Java-фреймворків, таких як: Spring, Quarkus та Micronaut [14]. Їх використання може впливати на продуктивність роботи системи, чи на час її розробки. Проте меншою мірою на економію ресурсів, оскільки сам фреймворк для роботи не потребує багато ресурсів, у порівнянні з об'ємом пам'яті, необхідним для зберігання дескрипторів зображень.

Використання типів і структур даних безпосередньо впливає на використання RAM. Зокрема, через створення непотрібних об'єктів в циклах [15], використання класів-обгорток замість примітивів та колекцій замість масивів [16], реалізація може потребувати більше ресурсів.

Внутрішня реалізація різних віртуальних машин (JVM) також може мати вплив [17]. Більшість з них мало чим відрізняються від реалізацій від офіційного вендора – Oracle, і випускаються під власним брендом через ліцензійні обмеження. Проте GraalVM вносить велику кількість додаткових опцій [18]. Зокрема, компіляція в нативний код та Polyglot, який дозволяє об'єднувати декілька мов в одному виконуваному файлі. Вона є предметом чисельних досліджень, які перевіряють ці функції [19-21]. Також важливо використовувати останні версії інструментів для розробників (JDK), адже вони пропонують більше реалізованих внутрішніх оптимізацій.

Формулювання мети і завдань дослідження. Сучасні системи пошуку зображень на основі вмісту зазвичай зберігають векторні дескриптори в оперативній пам'яті, що призводить до високого споживання RAM, особливо в системах, які оперують мільйонами дескрипторів. Мінімізація обсягу пам'яті, необхідного для роботи CBIR-систем, є актуальною задачею в умовах обробки великих

обсягів даних. Це дозволяє зменшити ресурсні витрати на робочих станціях і знизити фінансові витрати при розміщенні систем у хмарному середовищі.

У статті розглядається проблема надмірного використання оперативної пам'яті в CBIR-системах, реалізованих мовою Java. Для її вирішення запропоновано формалізований метод, що поєднує оптимізацію структур даних із нативною компіляцією застосунку за допомогою GraalVM.

Метою дослідження є розробка та експериментальна перевірка ефективності запропонованого методу на прикладі CBIR-системи, побудованої за моделлю Multidimensional Cube. Особливу увагу приділено оцінці впливу оптимізацій на використання оперативної пам'яті та продуктивність виконання. Розглянуті існуючі оптимізації не були перевірені у контексті CBIR-систем.

Для досягнення мети необхідно виконати наступні задачі дослідження:

1. Дослідити вплив вибору типів і структур даних для зберігання дескрипторів – зокрема, примітивних типів, класів-обгорток, масивів і колекцій.
2. Оцінити ефективність застосування нативної компіляції (тобто компіляції у виконуваний файл, специфічний для платформи) вихідного коду з використанням GraalVM.
3. Побудувати аналітичні оцінки використання пам'яті для різних варіантів реалізації та зіставити їх з результатами профілювання.

Запропоновані оптимізації безпосередньо взаємодіють із базовими механізмами роботи з пам'яттю та виконанням програмного коду, що дозволяє віднести їх до категорії низькорівневих оптимізацій.

Обмеження дослідження: оптимізації не повинні зменшувати гнучкість коду або ускладнювати його підтримку і масштабування. Зниження точності пошуку також є неприйнятним. Допускається збільшення накладних витрат на етапі компіляції, але не під час виконання програми.

Практичний внесок полягає у виявленні оптимізацій, що дозволяють зменшити використання оперативної пам'яті. Оптимізація вважається успішною, якщо зменшує використання RAM щонайменше на 10%. Отримані результати можуть бути корисними для інших моделей CBIR, а також у реалізаціях іншими мовами програмування, які мають подібні типи та структури даних і мають підтримку з боку GraalVM.

Наукова новизна запропонованого методу полягає у:

1. Формальному визначенні комбінованого підходу до низькорівневої оптимізації програмного забезпечення для CBIR-систем.
2. Теоретичному моделюванні та емпіричній перевірці використання оперативної пам'яті за різних конфігурацій структур даних.
3. Демонстрації практичного застосування GraalVM у сценаріях пошуку великого масштабу з вимірюваними перевагами щодо використання пам'яті та продуктивності.

Запропонований метод. Пропонується формалізований метод для зменшення використання оперативної пам'яті системами CBIR, реалізованими мовою програмування Java. Може застосовуватися у CBIR-системах, у яких векторні дескриптори зображень зберігаються в RAM, наприклад модель MDC. Або в системах інших предметних областей, де поширене векторне представлення даних. Цей метод базується на комбінації двох підходів:

1. Оптимізації використання типів і структур даних, що використовуються для представлення дескрипторів.
2. Компіляції програмного коду у нативний виконуваний файл за допомогою GraalVM.

В MDC та подібних CBIR-моделях, які зберігають дескриптори під час своєї роботи в RAM, більшу частину пам'яті займають саме дескриптори. В MDC дескриптор складається з:

1. Ідентифікатора зображення.
2. Оригінального вектора, що складається з дробових чисел та має довжину рівну степеню двійки.
3. Агрегованого вектора такого ж формату меншої довжини рівної степеню двійки. Цей вектор визначає комірку MDC, в якій буде розміщуватися дескриптор.

Формально це можна представити наступним чином. Нехай у моделі пошукової системи є N дескрипторів, тоді обсяг RAM, необхідний для її зберігання обчислюється наступним чином:

$$M = N \times D + F, D = (id + V + A), V \in \mathbb{R}^n, A \in \mathbb{R}^r \quad (1)$$

де M – загальний обсяг пам'яті, N – кількість дескрипторів, D – обсяг пам'яті, що займає дескриптор, F – обсяг пам'яті для зберігання службових об'єктів фреймворку, який

використовується для реалізації, *id* – пам'ять для зберігання ідентифікатора зображення, *V* – пам'ять для зберігання вектора дескриптора, *n* – натуральне число, ступень двійки, *A* – пам'ять для зберігання агрегованого вектора дескриптора, *r* – натуральне число, ступень двійки.

Саме *D* з формули (1) є основним предметом оптимізації цієї роботи і потребує детально оцінки. За допомогою використання більш ефективних типів і структур даних можна досягти оптимізації, як безпосередньо, так і завдяки уникненню непотрібних операцій приведення типів boxing/unboxing.

Другий етап – це компіляція програмного коду в нативний виконуваний файл за допомогою GraalVM. Цей крок дозволяє додатково зменшити обсяг пам'яті, зокрема завдяки:

1. Детальному аналізу і вилученню коду і залежностей, що не використовуються.

2. Створення нативного бінарного виконуваного файлу, який використовує найбільш ефективні типи і структури даних, які доступні для конкретної платформи.

Компіляція в нативний код охоплює всі компоненти з формули (1), включно з фреймворком, логікою обробки та допоміжними структурами. Таким чином, навіть після оптимізації дескрипторів на першому етапі, другий крок дозволяє досягти додаткового скорочення споживання пам'яті.

Ефективність цього кроку можна оцінити наступною формулою:

$$\Delta RAM = RAM_{java} - RAM_{native}, \quad (2)$$

де ΔRAM – різниця у обсязі пам'яті, яку вдалося заощадити під час виконання нативного коду, RAM_{java} – обсяг пам'яті, що використовується під час виконання після класичної компіляції, RAM_{native} – обсяг пам'яті, що використовується під час виконання після нативно компіляції.

Аналіз типів і структур даних. Ідентифікатор зображення, що міститься у дескрипторах має тип даних String. В Java він має реалізовані внутрішні оптимізації і займає об'єм пам'яті відповідно до вмісту. Якщо ідентифікатори зображення є латинськими символами в кодуванні Latin1, то кожен символ займає всього 1 байт. Додаткові поля об'єкту, такі як: заголовок об'єкту, заголовок масиву символів, поле, що вказує на кодування та вирівнювання розміру об'єкту до значення, кратного 8 в 64-бітних реалізаціях, може займати додатково до 40 байтів.

Вектори дескриптора мають фіксовану довжину, що дозволяє їх зберігати як в масивах – низькорівневих структурах даних, так і в списках – високорівневих колекціях з динамічним розміром. Від вибраної стратегії зберігання векторів і залежить більшою мірою ефективність використання пам'яті. В масивах можна зберігати як представлення дробових чисел у форматі примітивних типів даних double чи float так і у відповідних їм класах-обгортках Double, Float. В колекціях зберігання примітивів є неможливим через обмеження мови програмування.

Щодо до поставлених обмежень: використання масивів та примітивів замість колекцій і класів обгорток на впливає на гнучкість, підтримку чи масштабування, оскільки вектори дескриптора є сталими за розміром. Використання типу даних float і його відповідного класу-обгортки є неприпустимим, оскільки це призведе до зниження точності пошуку. Звісно, він займає 4 байти замість 8, менше у 2 рази, проте це може призвести до зменшення точності до 2 разів. Тому розглядаються варіанти використання масивів і колекцій з типом даних double і відповідного класу-обгортки. Використання типу float буде досліджено у іншій роботі.

При використанні масивів з примітивами типу double необхідний об'єм пам'яті для зберігання вектору дескриптора визначається наступною формулою:

$$size = ah + db \times N, \quad (3)$$

де *ah* – заголовок масиву (16 байт), *db* – кількість байтів, що займає тип double (8 байт), *N* – довжина масиву.

При використанні масивів з класом-обгорткою Double необхідний об'єм пам'яті для зберігання вектору дескриптора визначається наступною формулою:

$$size = ah + lb \times N + DB \times N, \quad (4)$$

де *ah* – заголовок масиву (16 байт), *lb* – кількість байтів, що займає посилання на об'єкт Double (4 байти), *N* – довжина масиву, *DB* – кількість байтів, що займає об'єкт Double (24 байти). *DB* з займає 24 байти, бо містить заголовок 12 байтів, саме значення типу double (8 байтів) та вирівнювання до степеню двійки (4 байти).

Тобто масив об'єктів-обгорток містить не самі об'єкти типу Double, а посилання на них. А об'єкти окремо розміщуються у кучі (heap). На відміну від цього, масив примітивів це просто послідовний фрагмент виділеної пам'яті. Через це масив з об'єктами-обгортками займає більше пам'яті і робота з ним є повільнішою. Для деяких операцій виконуються процеси boxing/unboxing

для приведення класу-обгортки до представлення у примітивному форматі і навпаки. Цей процес також негативно впливає на використання ресурсів.

Для використання колекцій з класом-обгорткою Double розглянемо реалізацію колекції ArrayList – списку з динамічним розміром, реалізованим на основі масиву. У цьому випадку необхідний об'єм пам'яті визначається наступною формулою:

$$size = al + la + ah + lb \times N + DB \times N, \quad (5)$$

де *al* – кількість байтів, що займає ArrayList (24 байти), *la* – кількість байт, що займає посилання на масив з посиланнями на об'єкти типу Double (8 байт), *ah* – заголовок масиву (16 байт), *lb* – кількість байтів, що займає посилання на об'єкт Double (4 байти), *N* – довжина масиву, *DB* – кількість байтів, що займає об'єкт Double (24 байти). Значення *al* потребує 24 байти, оскільки 16 байтів займає заголовок, 4 байти – поле з розміром масиву посилань і 4 байти службове поле, яке рахує кількість модифікацій. Додатково слід зауважити, що розмір колекції регулюється за певними правилами і не завжди співпадає з реально розміщеною в ній кількості елементів, що може викликати додаткові витрати пам'яті.

Вказані об'єми пам'яті можуть дещо відрізнятися в залежності від платформи (32-х чи 64-х бітної), версії JVM та її постачальника. Вказані підрахунки актуальні для 64-х бітної Amazon Corretto 17.0.14 та отримані за допомого вивчення реалізації вказаних типів та структур даних.

Компіляція в нативний код. Компіляція – це процес переведення програмного коду в код, доступний до виконання. Класичною для Java є компіляції в байт-код, який збирається у jar файл. Далі його виконує віртуальна машина за допомогою транслятора на конкретній операційній системі. Додатково у процесі виконання визначаються фрагменти програми, виконання яких можливо оптимізувати і виконати Just in Time (JIT) компіляцію.

Компіляція в нативний код є альтернативою до класичної. Для виконання нативної компіляції використовується віртуальна машина GraalVM. В цьому випадку прибирається проміжний етап з байт-кодом. І використовується підхід компіляції Ahead of Time (AOT). Це означає, що в момент компіляції код детально аналізується і у результуючий виконуваний файл будуть додані лише ті класи та ресурси, які безпосередньо використовуються. Будь-які інші ресурси чи метаінформація буде відкинута і має бути включена за допомогою додаткових конфігурацій. Це накладає певні обмеження на такі процеси як сериалізація, рефлексія чи динамічне завантаження коду. І надає наступні переваги: використання меншої кількості ресурсів під час запуску та роботи, швидкий старт, можливість миттєво обробляти пікове навантаження, малий розмір виконуваного файлу. Під час роботи нативного коду також працює збирач сміття, який дбає про утилізацію непотрібних об'єктів. Готовий виконуваний файл створюється для конкретної платформи. Для інших платформ чи операційних систем необхідно виконувати повторну компіляцію [22].

Для запуску нативної компіляції, реалізацію MDC необхідно підготувати, враховуючи вказані вище обмеження. Головним нюансом є правильна конфігурація класів, які піддаються сериалізації/десериалізації при завантаженні дескрипторів у систему, збереженні стану системи між запусками чи відображенні результатів пошуку. Для цього необхідно описати класи у спеціальних конфігураційних файлах, включаючи метаінформацію класу, таку як: поля, конструктори та методи з перерахуванням усіх параметрів та типів. Це дозволить вирішити вказані раніше обмеження AOT-компіляції.

Щодо поставлених в роботі обмежень: обмеження AOT-компіляції можуть бути вирішені описаним шляхом, є вірогідність, що деякі програмні бібліотеки не будуть працювати у цьому режимі, що може вплинути на гнучкість, підтримку і масштабування. У цьому випадку може використовуватися альтернативна програмна бібліотека чи адаптуватися логіка. У найгіршому випадку даний етап оптимізації на застосовується.

В перевагах нативної компіляції згадується зменшення кількості ресурсів при виконанні коду, тому цей підхід є обов'язковою опцією до розгляду в даному дослідженні. Дану оптимізацію можна вважати успішною, якщо нативний код буде споживати менше RAM та надавати таку ж продуктивність.

Експерименти і обговорення. Для проведення експериментів штучно сформовані 100 000 дескрипторів, описаного раніше формату: містять ідентифікатор файлу, оригінальний вектор властивостей зображення і додатковий компактний вектор. Довжина оригінального вектору дескриптора рівна 32, а довжина додаткового вектора – 4.

Програмна реалізація написана на Java 17. Для проведення експериментів використовується 64-бітна JVM Amazon Corretto 17.0.14 зі стандартно налаштованим GC G1. Для компіляції в нативний код використовується Oracle GraalVM 21.0.6 з включеним GC G1 замість стандартного Serial GC. В GraalVM використовується адаптований GC G1, отже його поведінка може дещо відрізнятися від класичної версії.

Для перевірки ефективності описаних у роботі оптимізацій проводяться такі експерименти:

1. Порівняння використання оперативної пам'яті для збереження векторів дескрипторів при використанні різних типів і структур даних: масиву з примітивами, масиву з класом-обгорткою Double та колекції ArrayList з класом-обгорткою Double. Інший код в програмному рішенні також реалізовано з використанням вибраних типів і структур даних.

2. Порівняння використання оперативної пам'яті при виконанні байт-коду звичайною віртуальною машиною та нативного коду, створеного за допомогою GraalVM. Дослідження проводиться для найкращого з варіантів, визначеного у пункті 1.

Експерименти проводяться на комп'ютері з такими характеристиками: SSD 512 ГБ, 16 ГБ RAM LPDDR3-2133 MHz, 4-ядерний процесор Intel i5 8250U 3.4 ГГц, 8 потоків, операційна система Manjaro Linux 24.1.1 з версією ядра 6.6.54-2, Docker 27.2.1.

В рамках першого експерименту було реалізовано програмний код пошукової системи на основі MDC з використанням типів і структур даних, вказаних в плані експерименту. Кожна з реалізацій була запущена, були завантажені дескриптори та за допомогою профайлеру (спеціального аналізатору пам'яті) встановлено, яку кількість оперативної пам'яті вони потребують.

У таблиці 1 наведена інформація про обсяг оперативної пам'яті, необхідний для зберігання 1 дескриптора та 100 000 дескрипторів. Додатково наведено інформацію середнього часу пошуку перших 100 найбільш схожих зображень для 100 вибраних випадковим чином дескрипторів зі сховища. Також додано теоретичну оцінку, отриману по розрахункам за формулами (3)-(5).

Таблиця 1. Порівняння ефективності використання різник типів і структур даних для розміщення дескрипторів зображень в RAM

Структура та тип даних	1 дескриптор (теоретично)	1 дескриптор	100 000 дескрипторів	Середній час пошуку
Масиви з примітивами	400 Б	400 Б	40 МБ	0.00048 с
Масиви з об'єктами	1.12 кБ	1.12 кБ	112 МБ	0.00067 с
Списки з об'єктами	1.2 кБ	1.2 кБ	120 МБ	0.00087 с

Фактично отримані результати повністю відповідають тим, що отримані аналітично. Тобто правильність аналітичного підходу підтверджена. Очікувано, оптимізована версія з використанням масивів з примітивами є найефективнішою та переважає використання найбільш наївного варіанту використання списків з об'єктами у 3 рази в ефективності використання RAM. Це показує оптимізацію найважливішого компонента *D* з формули (1). Середній час пошуку демонструє ефективність використання масивів з примітивами не лише з точки зору використання пам'яті, а і з точки зору швидкості виконання операцій. При роботі з класами обгортками через виконання операція boxing/unboxing продуктивність також є гіршою, ніж при роботі з примітивами.

Далі, для реалізації, що використовує масиви з примітивами, було виконано компіляцію в нативний код за допомогою GraalVM. Через наявні обмеження в програмному коді була замінена бібліотека для формування Excel файлів, що наявна у програмному рішенні. Інші складнощі під час нативної компіляції не виникали.

Через відсутність можливості запустити профайлер при виконанні нативного коду, порівняння виконувалося наступним чином – створено Docker образи для коду скомпільованого класичним і нативним способом. На основі образів були запущені Docker контейнери, в яких працювала система пошуку з MDC. В MDC було завантажено 1 мільйон дескрипторів, згенерованих випадковим чином. Для кожного контейнеру оцінювалися характеристики використання RAM: після запуску пошукової системи, після завантаження дескрипторів, після запуску з завантаженими дескрипторами та після виконання навантажувального тестування.

Навантажувальне тестування відбувалося у 3 етапи за допомогою утиліти Locust [23]: 1) 100 користувачів робили запити протягом 10 хвилин; 2) 1 000 користувачів робили запити протягом 20

хвилин; 3) 1 000 користувачів робили запити протягом 60 хвилин. Запити від віртуальних користувачів посилалися паралельно, між запитами від одного користувача робилася пауза 1-2 секунди. Усі запити були на пошук 100 найбільш схожих зображень на 1 вибране з наявних випадковим чином. Таке навантаження емулює хвилі напливу користувачів на систему і дозволяє аналізувати, як вона себе поводить під реальним навантаженням.

Отримані результати наведено у таблиці 2. Додатково наведено порахований за формулою (2) результат від застосування цієї оптимізації. За формулою (1) об'єм пам'яті, необхідний для дескрипторів становить 400 МБ.

Таблиця 2. Порівняння використання RAM при виконанні Java коду компільованого класичним і нативним варіантами

Метрика	Класична компіляція	Нативна компіляція	Різниця за формулою (2)
Використання RAM після першого запуску без дескрипторів	281.9 МБ	76.23 МБ	205.67 МБ
Використання RAM після завантаження 1 мільйону дескрипторів	2.063 ГБ	1.017 ГБ	1.046 ГБ
Використання RAM після запуску з 1 мільйоном дескрипторів	814 МБ	654 МБ	160 МБ
Використання RAM після навантажувального тестування, 100 користувачів протягом 10 хвилин	1.579 ГБ	1.734 ГБ	-0.155 ГБ
Використання RAM після навантажувального тестування, 1000 користувачів протягом 20 хвилин	1.702 ГБ	1.743 ГБ	-0.041 ГБ
Використання RAM після навантажувального тестування, 1000 користувачів протягом 60 хвилин	1.828 ГБ	1.787 ГБ	0.041 ГБ

Отримані результати показують, що у більшості сценаріїв використання RAM нативним кодом є ефективнішим від 20 до 73 відсотків у процесах не пов'язаних з виконанням пошуку. При виконанні пошуку під час навантажувального тестування видно різницю в роботі GC: класичного і адаптованого для нативного коду. Споживання RAM при класичному виконанні збільшується більш лінійно, однак в результаті споживання RAM при класичному виконанні є незначно більшим. Додатково у формулу (1) можна додати ефективність GC, як один з параметрів для точнішої оцінки ресурсів. Бо, як видно з результатів, його вплив на результат є досить суттєвим.

Додатково було оцінено час, необхідний для: старту системи без дескрипторів, з дескрипторами, час завантаження дескрипторів, середній час пошуку (як у попередньому експерименті) та пропускну можливість системи під час навантажувального тестування. Результати наведено у Таблиці 3.

Таблиця 3. Порівняння ефективності при виконанні Java коду компільованого в класичний і нативний варіанти

Метрика	Класична компіляція	Нативна компіляція
Час старту системи без дескрипторів	3.937 с	0.378 с
Час завантаження 1 мільйону дескрипторів	4.433 с	3.497 с
Час старту системи з 1 мільйоном дескрипторів	5.835 с	2.597 с

Середній час пошуку топ-100 найподібніших зображень	0.00046 с	0.00056 с
Кількість оброблених запитів протягом навантажувального тестування	2 862 953	2 862 036

Усі операції, крім пошуку виконуються швидше у нативно компільованому коді. Час, витрачений на пошук є трохи гіршим при нативній компіляції, проте це не сильно впливає на реальне використання системи, оскільки в результаті нативним рішенням було оброблено всього на 917 запитів менше (що становить різницю менше 1%), а сумарно було оброблено майже 3 мільйони запитів.

Додатково порівняно вторинні метрики, пов'язані з виконанням нативної компіляції, такі як: час компіляції, пам'ять необхідна для компіляції, розмір результуючого виконуваного файлу. Результати наведено у Таблиці 4.

Таблиця 4. Порівняння метрик процесу компіляції нативної і класичної версії

Метрика	Нативна компіляція	Класична компіляція
Час компіляції	6 хв 5 с	1 хв 30 с
Використання RAM під час компіляції	7.3 ГБ	700 МБ
Розмір виконуваного файлу	235.6 МБ	89 МБ

Нативна компіляція вимагає більше часу та оперативної пам'яті, навантаження на процесор в обох випадках було зафіксовано наближеним до 100%, а розмір фінального виконуваного файлу є більшим у 2.67 рази. Однак компіляцію потрібно виконувати лише при змінах вихідного коду чи цільової платформи, тому цей недолік не можна вважати суттєвим.

Висновки. Було представлено формалізований метод оптимізації використання оперативної пам'яті в системах пошуку зображень на основі вмісту. На прикладі системи пошуку, побудованої на основі моделі MDC та реалізованої мовою програмування Java. Запропонований метод є комбінованим і складається з двох етапів: оптимізації використання типів і структур даних, що використовуються для представлення дескрипторів; компіляції програмного коду у нативний виконуваний файл за допомогою GraalVM. Окрім завдань, пов'язаних з розробкою вказаних етапів, додатково було проведено експериментальну перевірку результатів, отриманих аналітично. Даний метод можна широко застосовувати для аналізу та оптимізації CBIR-систем та систем з інших предметних галузей, де широко використовуються дані у векторному представленні.

На етапі оптимізації використання типів і структур даних досліджувалося використання високорівневої структури ArrayList та типу даних Double та низькорівневих масивів та примітивів типу double. Обчислено та експериментально підтверджено, що використання згаданих низькорівневих компонентів мови програмування може зменшити об'єм необхідної оперативної пам'яті для зберігання дескрипторів до 3 разів. Дана оптимізація потребує незначної зміни коду реалізації і не має негативного впливу ні на розробку програмного коду, ні на підтримку чи його розширення.

Використання нативної компіляції вихідного коду в бінарний виконуваний файл замість байт-коду, зібраного у jar файл потребує більше у 10 разів ресурсів і у 4 рази часу під час самого процесу компіляції. В більшості випадків не має негативного впливу на розробку, підтримку чи розширення коду. Іноді може вимагати адаптацій у програмному коді чи заміни деяких бібліотек, що може призвести до неможливості застосування цієї оптимізації. При роботі програми спостерігається більш швидкий запуск системи, зменшене споживання оперативної пам'яті від 20 до 73 відсотків у процесах не пов'язаних з виконанням пошуку та приблизно таку ж витрату оперативної пам'яті при виконанні пошуку. Показники продуктивності пошуку у MDC у порівнянні зі звичайною компіляцією є гіршими менше ніж на 1% при порівнянні пропускної можливості запитів. Тому цю оптимізацію також можна вважати позитивною.

Навіть не дивлячись на винятки застосування другого етапу оптимізації, сумарний ефект від запропонованих оптимізацій, або лише від першого етапу оптимізацій, дає перевагу більш ніж на 10%, які були визначені у постановці завдання. Можна вважати, що усі поставлені завдання виконані, отримані результати є задовільними. Практично, застосування досліджених оптимізацій має позитивний ефект та рекомендується до впровадження у програмні рішення.

Подальшим кроком дослідження є вивчення впливу використання типу даних float замість double для збереження властивостей дескриптора у контексті якості, швидкості та необхідних ресурсів для процесу пошуку. Та створення розширеної версії формули оцінки обсягу необхідної пам'яті, що враховує ефективність збирача сміття.

Список бібліографічного опису

1.Wang Y.-C., Yang T.-T., Wang H.-W., Yan B.-C., Chen B. AVATAR: Robust Voice Search Engine Leveraging Autoregressive Document Retrieval and Contrastive Learning. *2023 Asia Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. Taipei, Taiwan, 2023. С. 2331–2335. DOI: 10.1109/apsipaasc58517.2023.10317360.
2.Li X., Yang J., Ma J. Recent developments of content-based image retrieval (CBIR). *Neurocomputing*. 2021. Т. 452. С. 675–689. DOI: 10.1016/j.neucom.2020.07.139.
3.Chung I.K.Y., Tran M., Nussinovitch E. Scaling Cross-Domain Content-Based Image Retrieval for E-commerce Snap and Search Application. 2022. DOI: 10.48550/arXiv.2204.11593.
4.Gupta D., Loane R., Gayen S., Demner-Fushman D. Medical image retrieval via nearest neighbor search on pre-trained image features. *Knowledge-Based Systems*. 2023. Т. 278. DOI: 10.1016/j.knosys.2023.110907.
5.Kumar N., How Many Google Searches Per Day. *DemandSage*. 2025. URL: https://www.demandsage.com/google-search-statistics (дата звернення: 29.03.2025).
6. Alsmadi M.K. Content-Based Image Retrieval Using Color, Shape and Texture Descriptors and Features. *Arabian Journal for Science and Engineering*. 2020. Т. 45, №4. С. 3317–3330. DOI: 10.1007/s13369-020-04384-y.
7.Deochake S., Channapattan V., Steelman G., BigBird: Big Data Storage and Analytics at Scale in Hybrid Cloud. 2022. DOI: 10.48550/arXiv.2203.11472.
8.Singhal S., Gupta N., Berwal P., Naveed Q. N., Lasisi A., Wodajo A.W. Energy Efficient Resource Allocation in Cloud Environment Using Metaheuristic Algorithm. *IEEE Access*. 2023. Т. 11. С. 126135–126146. DOI: 10.1109/access.2023.3330434.
9.Liu M., Pan L., Liu S. Cost Optimization for Cloud Storage from User Perspectives: Recent Advances, Taxonomy, and Survey. *ACM Computing Surveys*. 2023. Т. 55. С. 1–37. DOI: 10.1145/3582883.
10. Jégou H., Douze M., Schmid C. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence.* 2011. Т. 33, №. 1. С. 117–128. DOI: 10.1109/tpami.2010.57.
11. Danylenko S., Smelyakov S. Development of a multidimensional data model for efficient content-based image retrieval in big data storage. *Radioelectronic and Computer Systems*. 2025. Т. 2025, №1. С. 137–152. DOI: 10.32620/reks.2025.1.10.
12. Bandari S.S.G. Mitigating Critical Challenges in Java Production Environments: Memory Leaks, Dependency Conflicts, and Performance Optimization in Enterprise Systems. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*. 2024. Т. 10, №6. С. 2492–2499. DOI: 10.32628/cseit2425481.
13. Cai Z., Blackburn S.M., Bond M.D., Maas M., Distilling the Real Cost of Production Garbage Collectors. *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Singapore, Singapore, 2022. С. 46–57. DOI: 10.1109/ispass55109.2022.00005.
14. Wyciślik Ł., Latusik Ł., Kamińska A.M. A Comparative Assessment of JVM Frameworks to Develop Microservices. Applied Sciences. 2023. Т. 13, №3. DOI: 10.3390/app13031343.
15. Kumar Joshi P. Optimizing Web Applications Performance with Java: Best Practices. International Journal of Science and Research (IJSR). 2020. Т. 9, №9. С. 1649–1655. DOI: 10.21275/sr20921115232.
16. Makor L., Kloibhofer S., Hofer P., Leopoldseder D., Mössenböck H. Automated Profile-Guided Replacement of Data Structures to Reduce Memory Allocation. *The Art, Science, and Engineering of Programming*. 2025. Т. 10. DOI: 10.22152/programming-journal.org/2025/10/3.
17. Duda K. Comparing performance of JVM implementations. *Fine Constant.* 2020. URL: https://www.fineconstant.com/posts/comparing-jvm-performance/ (дата звернення: 29.03.2025).
18. Oracle and/or its affiliates. *GraalVM*. URL: https://www.graalvm.org/ (дата звернення:29.03.2025).
19. Šipek M., Muharemagić D., Mihaljević B., Radovan A. Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus. *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO).* Opatija, Croatia, 2020. С. 1746–1751, DOI: 10.23919/mipro48935.2020.9245290.
20. Šipek M., Mihaljević B. Radovan A. Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM. *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. Opatija, Croatia, 2019. С. 1671–1676. DOI: 10.23919/mipro.2019.8756917.
21. Danylenko S. Using GraalVM in a Real-world Scenario: Techstack's Experience. *Blog*. 2024. URL: https://tech-stack.com/blog/using-graalvm-in-a-real-world-scenario-techstacks-experience/ (дата звернення: 29.03.2025).
22. Oracle and/or its affiliates. *Native Image*. URL: https://www.graalvm.org/latest/reference-manual/native-image/ (дата звернення: 01.04.2025).
23. Locust.io. A Modern Load Testing Framework. URL: https://locust.io/ (дата звернення: 03.04.2025).

References

1.Wang Y.-C., Yang T.-T., Wang H.-W., Yan B.-C., Chen B. AVATAR: Robust Voice Search Engine Leveraging Autoregressive Document Retrieval and Contrastive Learning*. 2023 Asia Pacific Signal and Information Processing*

Association Annual Summit and Conference (APSIPA ASC). Taipei, Taiwan, 2023. P. 2331–2335. DOI: 10.1109/apsipaasc58517.2023.10317360.
2.Li X., Yang J., Ma J. Recent developments of content-based image retrieval (CBIR). *Neurocomputing*. 2021. Vol. 452. P. 675–689. DOI: 10.1016/j.neucom.2020.07.139.
3.Chung I.K.Y., Tran M., Nussinovitch E. Scaling Cross-Domain Content-Based Image Retrieval for E-commerce Snap and Search Application. 2022. DOI: 10.48550/arXiv.2204.11593.
4.Gupta D., Loane R., Gayen S., Demner-Fushman D. Medical image retrieval via nearest neighbor search on pre-trained image features. *Knowledge-Based Systems*. 2023. Vol. 278. DOI: 10.1016/j.knosys.2023.110907.
5.Kumar N., How Many Google Searches Per Day. *DemandSage*. 2025. URL: https://www.demandsage.com/google-search-statistics (access date: 29.03.2025).
6. Alsmadi M.K. Content-Based Image Retrieval Using Color, Shape and Texture Descriptors and Features. *Arabian Journal for Science and Engineering*. 2020. Vol. 45, №4. P. 3317–3330. DOI: 10.1007/s13369-020-04384-y.
7.Deochake S., Channapattan V., Steelman G., BigBird: Big Data Storage and Analytics at Scale in Hybrid Cloud. 2022. DOI: 10.48550/arXiv.2203.11472.
8.Singhal S., Gupta N., Berwal P., Naveed Q. N., Lasisi A., Wodajo A.W. Energy Efficient Resource Allocation in Cloud Environment Using Metaheuristic *Algorithm. IEEE Access*. 2023. Vol. 11. P. 126135–126146. DOI: 10.1109/access.2023.3330434.
9.Liu M., Pan L., Liu S. Cost Optimization for Cloud Storage from User Perspectives: Recent Advances, Taxonomy, and Survey. *ACM Computing Surveys*. 2023. Vol. 55. P. 1–37. DOI: 10.1145/3582883.
10. Jégou H., Douze M., Schmid C. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2011. Vol. 33, №. 1. P. 117–128. DOI: 10.1109/tpami.2010.57.
11. Danylenko S., Smelyakov S. Development of a multidimensional data model for efficient content-based image retrieval in big data storage. Radioelectronic and Computer Systems. 2025. Vol. 2025, №1. P. 137–152. DOI: 10.32620/reks.2025.1.10.
12. Bandari S.S.G. Mitigating Critical Challenges in Java Production Environments: Memory Leaks, Dependency Conflicts, and Performance Optimization in Enterprise Systems. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*. 2024. Vol. 10, №6. P. 2492–2499. DOI: 10.32628/cseit2425481.
13. Cai Z., Blackburn S.M., Bond M.D., Maas M., Distilling the Real Cost of Production Garbage Collectors. *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Singapore, Singapore, 2022. P. 46–57. DOI: 10.1109/ispass55109.2022.00005.
14. Wyciślik Ł., Latusik Ł., Kamińska A.M. A Comparative Assessment of JVM Frameworks to Develop Microservices. *Applied Sciences*. 2023. Vol. 13, №3. DOI: 10.3390/app13031343.
15. Kumar Joshi P. Optimizing Web Applications Performance with Java: Best Practices. *International Journal of Science and Research (IJSR)*. 2020. Vol. 9, №9. P. 1649–1655. DOI: 10.21275/sr20921115232.
16. Makor L., Kloibhofer S., Hofer P., Leopoldseder D., Mössenböck H., Automated Profile-Guided Replacement of Data Structures to Reduce Memory Allocation. *The Art, Science, and Engineering of Programming*. 2025. Vol. 10. DOI: 10.22152/programming-journal.org/2025/10/3.
17. Duda K. Comparing performance of JVM implementations. *Fine Constant*. 2020. URL: https://www.fineconstant.com/posts/comparing-jvm-performance/ (access date: 29.03.2025).
18. Oracle and/or its affiliates. *GraalVM*. URL: https://www.graalvm.org/ (access date: 29.03.2025).
19. Šipek M., Muharemagić D., Mihaljević B., Radovan A. Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus. *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. Opatija, Croatia, 2020. P. 1746–1751, DOI: 10.23919/mipro48935.2020.9245290.
20. Šipek M., Mihaljević B. Radovan A. Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM. *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO).* Opatija, Croatia, 2019. P. 1671–1676, DOI: 10.23919/mipro.2019.8756917.
21. Danylenko S. Using GraalVM in a Real-world Scenario: Techstack's Experience. *Blog*. 2024. URL: https://tech-stack.com/blog/using-graalvm-in-a-real-world-scenario-techstacks-experience/ (access date: 29.03.2025).
22. Oracle and/or its affiliates. *Native Image*. URL: https://www.graalvm.org/latest/reference-manual/native-image/ (access date: 01.04.2025).
23. Locust.io. A Modern Load Testing Framework. URL: https://locust.io/ (access date: 03.04.2025).