Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інфокомунікацій
(повна назва)
Кафедра Інфокомунікаційної інженерії імені В.В. Поповського
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Аналіз продуктивності методів хешування для пошуку подібності на вбудованих даних
(тема)

Виконав:
студент 2 курсу, групи АМСЗІм-20-1
Сахаров М. Г.
(прізвище, ініціали)

Спеціальність: 125 Кібербезпека
(код і повна назва спеціальності)
Тип програми: освітньо-наукова
(освітньо-професійна або освітньо-наукова)
Освітня програма: Адміністративний менеджмент у сфері захисту інформації
(повна назва освітньої програми)

Керівник: професор кафедри ІКІ ім. В.В. Поповського
Радівілова Т. А.
(посада, прізвище, ініціали)

2022р.

Харківський національний університет радіоелектроніки

Факультет Інфокомунікацій
(повна назва)
Кафедра Інфокомунікаційної інженерії імені В.В. Поповського
(повна назва)
Рівень вищої освіти другий (магістерський)
Спеціальність 125 Кібербезпека
(код і повна назва)
Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)
Освітня програма Адміністративний менеджмент у сфері захисту інформації
(повна назва)

ЗАТВЕРДЖУЮ
Зав. кафедри_____
(підпис)
«____» _____2022р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента Сахарова Марка Геннадійовича
(прізвище, ім'я, по батькові)

1. Тема роботи: Аналіз продуктивності методів хешування для пошуку подібності на вбудованих даних
затверджена наказом по університету від «24» березня 2022р. №410Ст.
2. Термін подання студентом роботи до екзаменаційної комісії 29.05.2022р.
3. Вихідні дані до роботи: математичні методи експертних оцінок, методи хешування MD5. SHA-1, SHA-256, методи шифрування DES та 3DES
4. Перелік питань, що потрібно опрацювати в роботі:
1) Що таке вбудовані дані
2) Оптимізація роботи з великим обсягом даних
3) Які існують методи хешування
4) Аналіз існуючих методів хешування
5) Пропозиції щодо майбутніх робіт

5. Перелік графічного матеріалу із зазначенням креслень, плакатів, комп'ютерних ілюстрацій: Демонстраційний матеріал у вигляді ppt-презентації;

6. Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		(підпис)	(дата)
Основна частина	професор кафедри ІКІ ім. В.В. Поповського Радівілова Т. А.		20.05.2022

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	24.03.2022	Виконано
2	Збір матеріалів для дослідження	15.04.2022	Виконано
3	Розробка 1 розділу	20.04.2022	Виконано
4	Розробка 2 розділу	25.04.2022	Виконано
5	Розробка 3 розділу	30.04.2022	Виконано
6	Розробка 4 розділу	05.05.2022	Виконано
7	Оформлення кваліфікаційної роботи	20.05.2022	Виконано

Дата видачі завдання 24 березня 2022 року

Студент _____ Сахаров М. Г.
(підпис) (прізвище, ініціали)

Керівник роботи _____ Радівілова Т. А.
(підпис) (посада, прізвище, ініціали)

Робота не містить відомостей заборонених до відкритого опублікування

Студент Марк САХАРОВ

Керівник роботи Тамара РАДІВІЛОВА

РЕФЕРАТ

Пояснювальна записка: 70 с., 25 рис., 25 табл., 23 джерел.

ВБУДОВАННІ ДАНІ, ХЕШУВАННЯ, ШИФРУВАННЯ, АНАЛІЗ, MD5, SHA-1, SHA-256, DES, 3DES.

Об’єкт дослідження – процес аналізу продуктивності роботи методів хешування.

Предмет дослідження – методи хешування.

Мета роботи – аналіз методів хешування для пошуку подібності на вбудованих даних.

Метод дослідження – криптографічний аналіз.

В кваліфікаційній роботі виконаний аналіз методів хешування для пошуку подібності на вбудованих даних.

ABSTRACT

The report contains 70 p., 25 fig., 25 tables, 23 sources.

EMBEDDED DATA, HASHING, ENCRYPTION, ANALYSIS, MD5, SHA-1, SHA-256, DES, 3DES.

The object of research is the process of analyzing the performance of hashing methods.

The subject of research - hashing methods.

The purpose of the work is to analyze hashing methods to find similarities in embedded data.

Research method – cryptography analysis.

In the qualification work the analysis of hashing methods for finding similarities on the embedded data is performed.

ЗМІСТ

Перелік скорочень, умовних позначень, символів, одиниць і термінів……... 8
Вступ…….……...……..... 9
1 Великомасштабна обробка даних……..…….…….…….…….…….... 10
1.1 Що таке вбудованість…….…….…….…….. 10
1.2 Обробка великомасштабних даних ……. 12
1.3 Приклади великомасштабних систем обробки даних……..……. 12
1.4 Оптимізатори для великомасштабної обробки потоків даних…….. 22
1.5 Сховище для великомасштабної обробки даних…..……...…… 26
2 Оптимізації для вбудовування та масивів даних…… 32
2.1 Оптимізоване керування вбудовуваннями……...……... 32
2.2 Багатомірне хешування……...…….. 34
2.3 Хешування, чутливе до місцевості…….. 35
2.4 Вивчення хешу...……..... 35
2.5 Постанова підконтрольної організації...……...… 40
3 Огляд проекту та реалізація прототипу……..…. 41
3.1 Огляд проекту……...… 41
3.2 Пропонований підхід……..... 42
3.3 Визначення проблеми……...… 43
3.4 Вивчення хеш-функції……...…... 44
3.5 Модельне навчання……...…… 47
3.6 Попередня обробка та векторізація вхідних наборів даних……...… 49
3.7 Метрики оцінювання……...…... 49
4 Аналіз методів хешування MD5, SHA-1 та SHA-256……. 51
4.1 Показники оцінки виконання……...…… 51
4.2 Результати експериментів……. 51
4.3 Метод хешування MD5……...…….. 52
4.4 Метод хешування SHA-1……...……... 62
4.5 Метод хешування SHA-256……...……... 64

Висновки……...……..…….. 68
Перелік джерел посилання..……...……...…….. 69

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ ТА ТЕРМІНІВ

МН – машинне навчання
ЦП – центральний процесор
ANN – approximate nearest neighbor
API – application programming interface
BSP – bulk-synchronous parallel
BSVD – big spatial vector data
CPU – central processing unit
CQL – contextual query language
DDL – data definition language
DES – data encryption standard
DFS – distributed file system
DML – data manipulation language
DSL – domain-specific language
EDFS – efficient distributed file system
GFS – global file system
GPFS – general parallel file system
HDFS – hadoop distributed file system
IR – intermediate representation
L2H – learning to hash
LSH – locality sensitive hashing
QFS – quantcastfs
RDD – resilient distributed datasets
RFS – ring file system
SHA – secure hash algorithm
SPE – stream processing engines
SQL – structured query language

ВСТУП

У сучасну епоху швидкість накопичення даних є експоненційною, що робить пошук відповідної інформації все більш складним. У такому сценарії високовимірний пошук подібності є популярним методом для вилучення релевантної інформації з великих обсягів даних або великих даних, і він додатково запускає різні завдання машинного навчання (МН), включаючи виявлення майже дублікатів і розпізнавання місцеположення. Однак, завдяки своїм характеристикам, великі дані ставлять перед додатками машинного навчання різноманітні проблеми, такі як дисбаланс високих класів, необхідність розробки функцій для підтримки гетерогенних даних і потреба в ефективних рішеннях для запитів над даними масиву.

Отже, разом неймовірною швидкістю накопичення даних, зростає і ймовірність колізій, подібності. Чи більше обсяг даних, тим більше часу займає процес пошуку подібності. Саме для цього виникає необхідність знаходження більш продуктивного методу хешування. Тому актуальність питання аналізу продуктивності методів хешування – неймовірна.

Саме тому метою цієї роботи є проведення аналізу методів хешування для пошуку подібності на вбудованих даних.

Для виконання поставленої мети, у першому розділі кваліфікаційної роботи був зроблений огляд в першу чергу питань: що таке вбудованість, обробка величезної кількості даних та ін.

У другому розділі були розглянути методи оптимізації даних.

У третьому розділі були розглянуті питання вивчення проблеми стосовно керування вбудованістю та використання хешу.

У четвертому розділі був проведений безпосередньо аналіз методів хешування: MD5, SHA-1, SHA-256.

Окремі результати роботи доповідались на міжнародній науковій конференції [1, 2].

1 ВЕЛИКОМАСШТАБНА ОБРОБКА ДАНИХ

1.1 Що таке вбудованість

Як відомо, вбудовуванням є переклад високорозмірних векторів або неевклідових даних у простір низької розмірності. У контексті текстового введення слова вбудовування здатні перетворювати кожну роботу в текст в окремий щільний безперервний вектор із заданою кількістю вимірів, зберігаючи при цьому їх семантичну інформацію за рахунок розміщення подібних введених даних близько один до одного в просторі вбудовування. Методи вбудовування тексту можна розділити на категорії, засновані на підрахунку, які ґрунтуються на моделі мішка слів, де порядок слів ігнорується, наприклад, на тематичні моделі та на основі прогнозів на основі моделей послідовності слів, де порядок слів ігнорується. слова враховуються, наприклад, нейронні моделі. Крім того, вбудовування можна також створювати для форматів вхідних даних, включаючи графіки, аудіо/відео тощо. Як видно на рис. 1.1, на якому візуалізовано реальні вставки, він зображує геометричні відносини, які фіксують семантичні відносини між країною та її столицею зліва та часом дієслова справа.

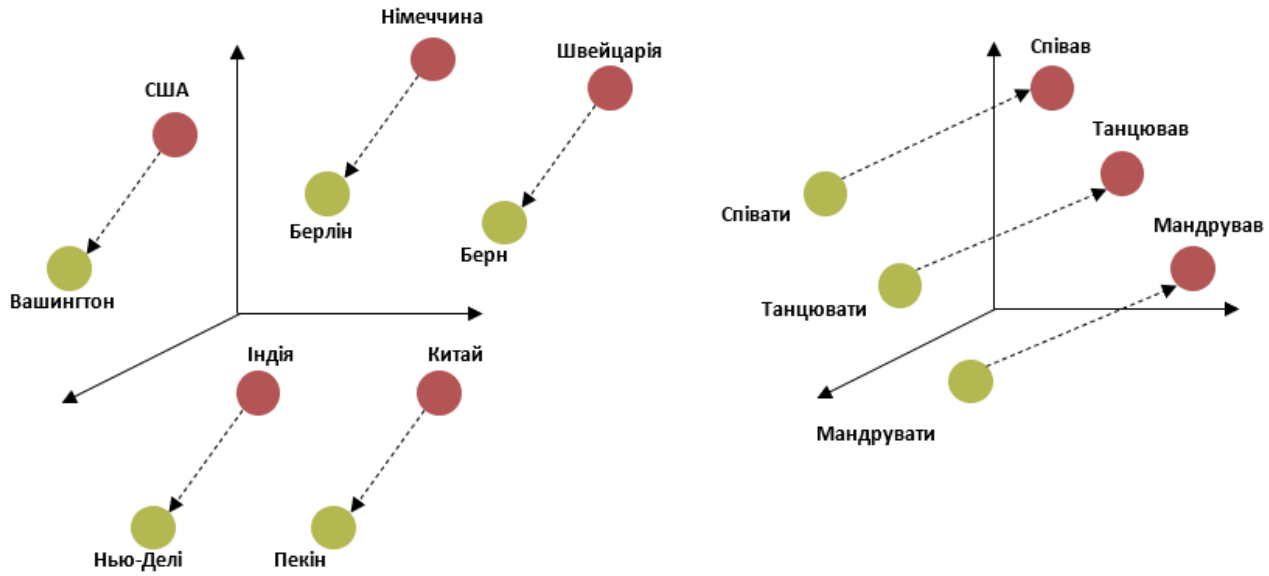

Рисунок 1.1 – Візуалізація вбудовування аналогій країни-столиці (зліва) та часу дієслова (праворуч)

Отже, вбудовування спрощує та ефективніше виконувати машинне навчання (МН) на відносно великих вхідних даних із можливістю навчання та повторного використання вбудовування як функцій у різних моделях для різних завдань. Простір вбудовування також є свого роду змістовним простором, який дає можливість системі МН виявляти семантичні шаблони, корисні для навчальних завдань. Крім того, простір вбудовування з його посиленням подібності може допомогти в задачах пошуку схожості, які є поширеними в системах пошуку інформації або рекомендаційних системах.

Останніми роками багато підходів до вбудовування різних наборів даних, з практикуючими спеціалістами, які зробили доступними попередньо навчені моделі, такі як Glove, Word2Vec та fastText..

Розглядаючи область застосування та вплив вбудовування спеціально для МН, на рис. 1.2 зображено різні проблеми, з якими стикаються програми МН у контексті чотирьох характеристик великих даних, а саме: обсягу, швидкості, різноманітності та правдивості.

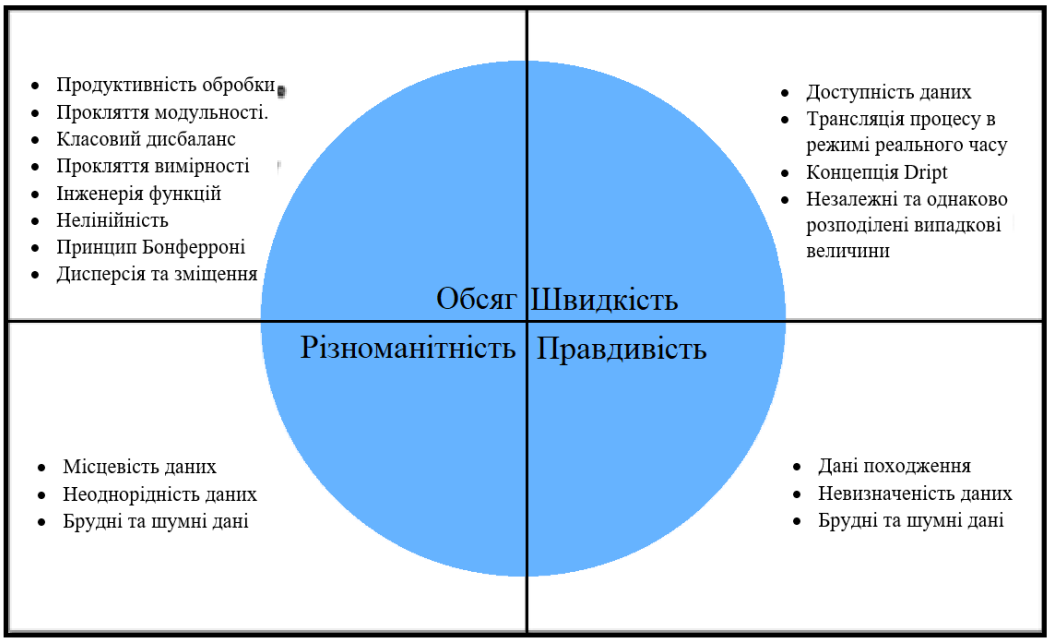

Рисунок 1.2 – Проблеми машинного навчання, пов’язані з характеристиками BigData

У такому сценарії вбудовування діє як техніка маніпулювання даними в конвеєрі аналізу даних , що допомагає пом’якшити прокляття розмірності та підвищити керованість даними за рахунок збереження їх семантичних зв’язків.

1.2 Обробка великомасштабних даних

У сучасну цифрову епоху світ переповнений величезним і вибуховим зростанням даних з різних джерел і в різних форматах, які також називають великими даними. Отже, великі дані ставлять перед своїми базовими структурами різні проблеми у великомасштабному зберіганні та обробці, що призвело до постійної еволюції у великомасштабних системах обробки даних. Великі дані самі по собі не корисні, якщо з ними не пов’язана утиліта. Тому широкомасштабна обробка таких даних для вилучення з них сенсу та цінності стає все більш важливою для надання корисності цим даним, таким чином слугуючи основою для прийняття рішень, керованих даними, візуалізуючи повну картину відповідної області програми. . Різні академічні та промислові сектори все більше залежать від знань, отриманих з Big Data, що робить їх ключовим ресурсом у сучасному сучасному світі, що робить інструменти Big Data вирішальними для оптимізованої та ефективної обробки цих великомасштабних даних. Масштабні системи обробки даних є майже синонімом інструментів Big Data. Їх можна розділити на різні системні рівні залежно від їхніх функцій і застосовності, а саме: рівень зберігання даних, рівень обробки даних, рівень запитів даних, рівень доступу до даних і рівень керування. Отже, підвищений попит на обробку та аналіз великомасштабних даних спричинив збільшення розвитку великомасштабних систем обробки даних у наукових та промислових середовищах.

1.3 Приклади великомасштабних систем обробки даних

Існує кілька великомасштабних систем обробки даних, призначених для надання різних функцій. Нижче наведено деякі з прикладів великомасштабних систем обробки даних.

Масово-синхронний паралельний (BSP). Масово-синхронна паралельна модель діє як міст між теоретичними та практичними аспектами реалізації паралельних обчислень. BSP не є ані програмною, ані апаратною моделлю, а лежить між ними, діючи як стандарт для відображення програм високого рівня на машини, максимізуючи ефективність паралельних обчислень. Модель BSP містить у собі три атрибути, а саме: ряд компонентів, які виконують функції пам'яті/обробки, маршрутизатор, який доставляє повідомлення між компонентами,

і механізм синхронізації між компонентами. Обчислення в моделі BSP складається з послідовності суперкроків, під час яких кожен компонент обробки виконує локальне обчислення з відправкою та отриманням повідомлень до та від інших компонентів. При обробці на кожному суперкроку локальні обчислення залежать лише від локальних даних, присутніх у пам'яті на початку кожного суперкроку з шаблоном зв'язку, відомим як h-відношення, що означає, що кожен компонент обробки може надсилати та отримувати максимум визначених h-повідомлення. Оскільки компоненти й маршрутизатор є різними об'єктами, це створює спрощену модель, яка розділяє задачі обчислення та зв'язку. Щоб забезпечити гарантії продуктивності, модель BSP використовує бар'єрну синхронізацію, яка забезпечує синхронізацію всіх компонентів обробки в певній точці, в кінці суперкроку. Крім того, щоб скоротити час очікування, механізм синхронізації можна вимкнути для підмножини компонентів, що призводить до певної асинхронної поведінки. Отже, модель BSP спрямована на досягнення гарантованої продуктивності при близькому до оптимального використання розподілених процесорів, що забезпечує високу пропускну здатність при паралельних обчисленнях. Як приклад програми, багаторівнева організація програм Pregel (запроваджена пізніше в їх Large-Scale Graph Processing) базується на моделі BSP.

MapReduce. MapReduce – це обчислювальний фреймворк для ефективної обробки великих наборів даних у паралельному розподіленому середовищі. На відміну від BSP, він заснований на спрощеній моделі обмеженого програмування, яка автоматично паралелізує обчислення, забезпечуючи ефективну відмовостійкість. У моделі програмування MapReduce користувачі можуть виразити обчислення, реалізуючи дві функції, а саме Map і Reduce. Враховуючи набір вхідних пар ключ/значення, функція Map обробляє вхідні дані, щоб створити проміжні пари ключ/значення як вхід до функції Reduce, яка потім об'єднує всі значення, пов'язані з одним ключем. У паралельному розподіленому середовищі виконання вхідні дані, які підлягають обробці, автоматично розподіляються на M розділів бібліотекою MapReduce з робочим навантаженням, розподіленим між набором робочих вузлів, які виконують завдання Map і Reduce, і всі керуються головним. Вихідні дані обробки MapReduce тоді доступні як набір вихідних файлів, що відповідають кожному завданню Reduce. Крім того, MapReduce реалізує ряд оптимізацій для мінімізації мережевого трафіку, включаючи планування з урахуванням місцевості. Отже, MapReduce як модель обмеженого програмування

абстрагує складнощі, пов’язані з великомасштабною паралельною розподіленою обробкою (наприклад, балансування навантаження, відмовостійкість), забезпечуючи таким чином простоту використання для вираження широкого спектру проблем.

Потік даних. Платформи обчислень для пакетно-орієнтованого потоку даних, такі як MapReduce, не забезпечують ефективно повторне використання даних проміжних результатів у кількох обчисленнях. Як наслідок, використання стійких розподілених наборів даних (RDD), запропонованих Zaharia та ін., або подібних концепцій, які зберігають дані в пам’яті та можуть відновлювати їх без реплікації шляхом відстеження графіка походження, що складається з операцій, що використовуються для побудови даних, стає більш оптимальною альтернативою. Механізми потоку даних є цією структурою, яка використовує такий обчислювальний графік, без вимог пакетної обробки та точок синхронізації, як у Map Reduce. Spark із внутрішнім використанням RDD є прикладом механізму потоку даних. У цьому прикладі кожен RDD є доступною лише для читання, розділеною колекцією записів із незмінною природою і може бути створена за допомогою детермінованих операцій над стабільними даними зберігання або з інших RDD. Користувачі можуть керувати аспектами RDD, а саме, розділенням і збереженням, що дозволяє специфікувати повторне використання сховища та оптимізувати стратегії розміщення. Таким чином, RDD, реалізовані в Spark, дозволяють користувачам кешувати попередні обчислення в пам’яті, завдяки чому вони добре підходять для ітеративних програм машинного навчання та інтерактивного аналізу даних. Spark надає функціональний програмний API на мові програмування Scala для маніпулювання RDD як розподілених абстракцій пам'яті з грубозернистими перетвореннями, такими як карта, фільтр і приєднання, а також надає підтримку інтерактивного запиту великих наборів даних з інтерпретатора Scala. Отже, Spark з RDD може ефективно виражати низку моделей кластерного програмування, а саме Pregel, HaLoop тощо, і стає ефективною моделлю програмування для пакетної аналітики. Однак Spark має понад 150 параметрів, які можна налаштувати, що робить пошук оптимальної конфігурації параметрів для додатків і кластерів складним завданням, щоб скоротити час обробки та підвищити загальну продуктивність. Розглядаючи налаштування параметрів, експериментальні дослідження забезпечують систематичний підхід до встановлення параметрів шляхом дослідження впливу деяких важливих

налаштованих параметрів, пов’язаних із перемішуванням, стисненням та серіалізацією, на продуктивність програм Spark.

Своєчасний потік даних. Своєчасний потік даних — це модель паралельного обчислення даних, яка розширює традиційну модель потоку даних, пов’язуючи кожну подію зв’язку з віртуальною міткою часу. Як приклад, Naiad представляє себе як єдину платформу для виконання додатків із паралельним циклічним потоком даних із збереженням стану в розподіленій установці з високою пропускною здатністю та низькою затримкою. Naiad розширює традиційний потік даних, щоб забезпечити підтримку інкрементного та ітераційного паралельного обчислення даних. Механізм віртуальних часових міток, що утворюють своєчасний потік даних у Naiad, сприяє ефективній та легкої координації в розподіленій установці. Ці логічні позначки часу розроблені на основі обмеженої циклічної структури, присутньої в графіках своєчасного потоку даних, в якій вершини орієнтованого графа організовані у можливі вкладені контексти циклу, що містять цикли в графі та пов’язані з трьома спеціальними вузлами, а саме вхідним, вихідним і зворотним зв’язком. Краї, що входять у контекст циклу, повинні проходити через вхідний вузол, а ті, що виходять, повинні проходити через вихідний вузол. Крім того, щонайменше один вузол зворотного зв’язку присутній у контексті циклу. Потім ці вершини діють і відповідно коригують часові позначки повідомлень, що проходять через них. Крім того, Naiad надає методи оптимізації для зменшення накладних витрат на зв’язок у розподіленій установці, а також для механізму відмовостійкості, що забезпечує послідовне відновлення у разі збоїв. Naiad пропонує гнучку структуру таким чином, що багато потужних моделей програмування можуть бути побудовані на його низькорівневих примітивах, що дозволяє виконувати різноманітні завдання, такі як потоковий аналіз даних, ітераційне машинне навчання та інтерактивний аналіз графів.

Потокове. Потокова передача пов’язана з обробкою та обробкою даних, які постійно отримуються з різних джерел, з додатковим аспектом, що запити часто відбуваються через обмежене вікно нещодавно переглянутих даних, а не над повними історичними даними. У порівнянні з обробкою пакетно-орієнтованих даних, потокові дані мають різні характеристики та мають ряд проблем, які вимагають спеціалізованих ефективних систем для їх обробки. Ці спеціалізовані системи, відомі як Stream Processing Engines (SPE), призначені для обробки потокових даних, коли вони надходять «на льоту», без необхідності їх зберігання.

Крім того, ці SPE повинні відповідати восьми вимогам обробки потоків, щоб мати можливість ефективно обробляти потоки даних великого обсягу з низькою затримкою і демонструвати різноманітні можливості застосування. Проект STREAM у Стенфорді є репрезентативною дослідницькою ініціативою, яка пропонує систему керування потоками даних (DSMS) для обробки безперервних потоків даних. STREAM, запропонований як DSMS загального призначення, використовує власну мову запитів, яка називається мовою безперервних запитів (CQL), яка розширює SQL, щоб бути адаптованим для безперервних запитів до потоків даних. Крім того, CQL, як мова запитів, достатньо виразний, щоб охопити динамічні характеристики потоків даних за допомогою складних запитів. Крім того, безперервні запити, зазначені в CQL, оцінюються шляхом їх перекладу на фізичний план запитів, який буде виконуватися DSDM. STREAM використовує декілька механізмів, включаючи усунення надмірності даних, оптимізоване планування оператора, щоб ефективно виконувати ці плани запитів і покращувати загальну продуктивність системи. Крім того, STREAM компромісує StreaMon для моніторингу та адаптивної обробки запитів для потоків даних із різними характеристиками. Крім того, STREAM також включає графічний запит і системний візуалізатор, щоб полегшити користувачам інтерактивну перевірку системи.

Storm — популярний проект для обробки потоків, який слугує системою розподіленої обробки потоків даних у режимі реального часу в Twitter. Storm — це масштабована та стійка система, яка здатна легко й ефективно аналізувати потоки соціальних мереж, як у Twitter, і підтримує рішення на основі даних у реальному часі. Ефективна підтримка потокової аналітики в реальному часі також вимагає ефективної системи управління для збору та доставки великих обсягів потокових даних. Apache Kafka підтримує цю вимогу, зосереджуючи увагу на управлінні сховищем та комунікації замість обробки запитів. Він служить розподіленою системою обміну повідомленнями для управління великими обсягами потокових даних за допомогою аналітики в пам’яті, полегшуючи переміщення великих даних надійним і масштабованим способом. Kafka забезпечує підтримку автономної та онлайн-аналітики, а також демонструє перевагу використання моделі споживання на основі витягування, яка дозволяє програмі споживати дані відповідно до власних потреб без перевантаження, максимізуючи таким чином загальну пропускну

здатність. Крім того, у Storm Kafka можна використовувати для вилучення даних із черг повідомлень, для створення потоку кортежів для їх подальшої обробки.

Пакет зустрічається з потоковим. Batch meets Streaming — це системи, розроблені для підтримки як пакетної, так і потокової обробки даних. Наприклад, Apache Flink — це система з відкритим вихідним кодом, що забезпечує уніфіковану модель єдиного виконання для пакетної та потокової обробки даних, яка може виражати та виконувати різні класи програм обробки даних у вигляді конвеєрних потоків даних, стійких до відмов. Розгортання, основний час виконання (система розподіленого потоку даних), API та бібліотеки є чотирма основними рівнями, які сприяють архітектурі Flink з усіма програмами, які в кінцевому підсумку компілюються у загальне представлення графа потоку даних. Flink забезпечує складний дуже гнучкий механізм вікон, який може обчислювати як ранні, приблизні, так і відкладені та точні результати в одній і тій же операції при обробці великомасштабного потоку даних. Крім того, він також підтримує різні поняття часу, забезпечуючи гнучкість у визначенні кореляції подій при обробці потоку даних. Flink також підтримує конкурентоспроможну оптимізовану пакетну обробку за рахунок включення спеціалізованих API, структур даних і алгоритмів. Зрештою, Flink як уніфікована платформа для обробки пакетних і потокових даних підтримує обробку поза порядком, ітераційну обробку, узгодженість завдяки підтримці одноразового стану та забезпечує результати з високою пропускною здатністю та низькою затримкою.

Гібриди. Вибір відповідного гібриду систем для великомасштабного робочого процесу обробки даних може бути корисним для досягнення високої пропускної здатності. За відсутності менеджера динамічного робочого процесу, проектування, впровадження або масштабування такого робочого процесу вимагає високого досвіду та часто є складним і трудомістким, головним чином через тісний зв’язок між інтерфейсними фреймворками та серверними механізмами виконання. У такому сценарії Musketeer є першим менеджером робочого процесу великих даних, який роз’єднує це зв’язок, забезпечуючи розширену гнучкість, полегшуючи користувачам написати робочий процес один раз і динамічно зіставляти його з багатьма різними системами. Зрештою, це робить проектування, впровадження та перенесення робочого процесу на інші системи набагато простішим та ефективнішим у порівнянні зі фіксованими відображеннями однієї системи. Відокремлена архітектура обробки даних Musketeer розділена на три рівні, а саме:

Front-End Frameworks, які користувачі використовують для визначення свого робочого процесу за допомогою високорівневих абстракцій, проміжне представлення (IR), яке динамічно перетворює та оптимізує вказаний робочий процес у загальну форму, прийнятну для передній і задній край, в ідеалі у вигляді орієнтованого ациклічного графа (DAG), і Back-End Execution Engines, які на основі користувача або автоматичного вибору виконують завдання, створені з DAG. Крім того, незважаючи на те, що Musketeer підтримує лише обмежені інтерфейсні фреймворки та серверні механізми виконання, цей підхід можна розширити, щоб підтримувати й інші, зрештою полегшуючи переважно неспеціалістам користувачам, які легко впроваджують гнучкі робочі процеси. Інший варіант оптимізатора робочого процесу, Weld, забезпечує загальне середовище виконання для оптимізації між бібліотеками в складних робочих процесах і компромісів IR на основі паралельних циклів і конструкції злиття результатів, яка називається конструкторами, яка може ефективно фіксувати перетворення циклу в порівнянні з IR, що використовується в Musketeer. Weld як програмний інтерфейс складається з трьох ключових компонентів для оптимізації аналітичних додатків, а саме, проміжного представлення, яке дозволяє бібліотекам виражати свою структуру паралельних обчислень, API середовища виконання, який збирає IR-код з різних бібліотек для відкладеного виконання, і компілятора. Бекенд, який компілює повну оптимізовану програму Weld IR для паралельного машинного коду, який буде виконуватися з даними програми в пам’яті. Таким чином, Weld забезпечує ефективну композицію гібридів бібліотек, що використовуються в додатках для аналізу даних, що дозволяє здійснювати наскрізну оптимізацію.

SQL на Hadoop. Системи SQL-on-BigData, як правило, діляться на дві категорії, а саме на власні системи на основі Hadoop, наприклад, Apache Hive та гібриди бази даних і Hadoop, наприклад, Hadapt, які інтегрують наявні інструменти з екосистемою Hadoop. Крім того, продукти на основі аналітичних пристроїв, наприклад, Oracle Exadata, мають роз’єми для систем зберігання великих даних, звідки вони витягують дані для обробки в своїх власних механізмах SQL. На рис. 1.3 влучно зображені різні відкриті та комерційні системи, доступні в ландшафті SQL-on-BigData.

Рисунок 1.3 – Системи, що порушують ландшафт SQL-on-BigData

SQL для обробки великих даних має велике значення, враховуючи його знайомство більшості розробників, а також простоту роботи з ним як інтуїтивно зрозумілою декларативною мовою високого рівня, що призводить до збільшення розробки переважно систем SQL-on-Hadoop. Ці системи усувають обмеження фреймворку MapReduce, які вимагають значного обсягу програмування для завдань, які можна виконати за допомогою простих команд SQL, що додатково додає його складності та обмеженої зручності для великомасштабної обробки даних. Деякі з цих систем включають Drill, HAWQ (Hadoop із запитом), Hive, Impala, Presto та Spark SQL. Ці системи корисні для спеціальних запитів та аналізу при обробці великомасштабних даних, як проаналізовано в дослідженні або Родрігес та ін., де виділено їх основні характеристики. У цьому дослідженні встановлено, що жоден інструмент не є оптимальним, але різні інструменти можна ефективно використовувати для різних вимог, наприклад, Hive стійкий до довгострокових запитів і ефективний для пакетної обробки, Impala і HAWQ можна краще використовувати для реального використання. аналіз часу завдяки їх швидкому часу відгуку, включаючи Spark, їх також можна краще використовувати для розширеної аналітики. Ефективність цих систем далі експериментально оцінюється в подальших дослідженнях авторів за допомогою тестів TPC-H і TPC-DS відповідно, при цьому обидві оцінки доводять той факт, що не існує єдиного розміру, який підходить для всіх SQL на Вимоги до обробки Hadoop. Крім того, в подальшому експериментальному дослідженні детально оцінюється продуктивність систем Apache Hive, Spark SQL, Apache Impala і PrestoDB,

використовуючи три різні тести, а саме TPC-H, TPC-DS і TPCx-BB, враховуючи різні сценарії застосування та характеристики робочого навантаження. Розглядаючи характеристики деяких систем SQL-on-Hadoop, які використовують фреймворк MapReduce для виконання запитів, Apache Hive є першою роботою, спрямованою на забезпечення масштабованого рішення сховища даних з відкритим вихідним кодом, побудованого на основі Hadoop з підтримкою запитів, виражених у мова запитів, подібна до SQL HiveQL. Будучи гнучким і розширюваним, HiveQL підтримує більшість SQL-подібних конструкцій запитів, а також оператори визначення даних (DDL) та маніпулювання (DMH), що полегшує запити, створення та маніпулювання даними. Запит HiveQL в основному компілюється в спрямований ациклічний графік (DAG) завдань MapReduce для виконання. Основні компоненти архітектури Hive включають зовнішні інтерфейси, що складаються з інтерфейсів програмування користувача та прикладних програм (API), сервер Thrift, Metastore як системний каталог, драйвер, який керує компіляцією, оптимізацією та виконанням оператора HiveQL, компілятор і Hadoop як механізм виконання. Ще одна система Apache Pig надає високорівневу мову сценаріїв Pig Latin, яка поєднує в собі аспекти високорівневих декларативних запитів, таких як SQL, і низькорівневого процедурного програмування, такого як MapReduce, для вираження великомасштабних завдань обробки даних. Програма Pig Latin, як правило, являє собою послідовність кроків, на кожному з яких здійснюється окреме перетворення даних. Що стосується виконання програми, фреймворк виконання Pig спочатку створює логічний план запиту з програми Pig Latin, який потім компілюється в серію завдань MapReduce для виконання переважно через Hadoop. Крім того, Pig також складається з власного середовища налагодження під назвою Pig Pen, що полегшує користувачам поетапне створення та налагодження своїх програм Pig Latin. Навпаки, існують також системи SQL-on-Hadoop, які для виконання завдань не покладаються на інші фреймворки, такі як MapReduce. Наприклад, Apache Impala — це механізм SQL-запитів із масивною паралельною обробкою (MPP) для даних, що зберігаються за допомогою Hadoop. Impala використовує архітектуру паралельної бази даних без спільного доступу на Hadoop за допомогою виконання запитів із використанням власних довготривалих демонів на кожному з HDFS DataNode замість того, щоб покладатися на фреймворк MapReduce. Існує головний демон процесу impalad, який містить планувальник запитів, координатор і механізм виконання. Зрештою, усі запити збираються в

конвеєрний план виконання. Розглядаючи експериментальну оцінку та порівняльний аналіз цих двох варіантів систем SQL-on-Hadoop, дослідження Floratou et al. забезпечує порівняння продуктивності Apache Hive і Apache Impala з використанням робочих навантажень, подібних до TPC-H і TPC-DS. У цьому дослідженні оцінюються значні переваги в продуктивності Impala над Hive, таким чином підкреслюються переваги використання архітектури бази даних без спільного доступу в порівнянні з середовищем виконання на основі MapReduce для аналітичних запитів SQL.

Apache Kudu — це новий механізм зберігання структурованих даних, який відіграє середину між швидким послідовним доступом і швидким довільним доступом. Він може працювати паралельно з існуючою інсталяцією HDFS і доступний за допомогою таких інструментів, як Impala, Spark і MapReduce. Отже, Kudu можна добре використовувати для швидкої аналітики над добре структурованими розподіленими даними через Hadoop.

Для цієї роботи я використав Apache Spark із запитами, подібними до SQL, що надаються інтерфейсом фрейму даних Spark.

Обробка великомасштабних графіків. Графіки великого розміру мають різні проблеми для їх ефективної обробки. Отже, ефективні обчислювальні моделі, що забезпечують конкурентоспроможну продуктивність, необхідні для обробки таких складних масивів даних графів великого розміру. Наприклад, Pregel як платформа розподіленого програмування від Google сприяє ефективній обробці великомасштабних графіків із збереженням стану з високою масштабованістю, продуктивністю та стійкістю до відмов. Pregel дозволяє реалізувати довільний алгоритм графа з введенням і виведенням програми Прегеля, в ідеалі є ізоморфними орієнтованими графами без обмежень, накладених на конкретний вибір формату файлу для цих графіків. Типове обчислення Прегеля складається з послідовності ітерацій, відомих як суперкроки, під час яких паралельно викликається визначена користувачем функція, що містить логіку обробки в кожній вершині. Зв'язок у всьому графі відбувається через його ребра з огляду на дизайн Прегеля для розріджених графів; з доставкою повідомлень асинхронно в пакетах, що зменшує затримку. Крім того, програми Pregel також за своєю суттю не мають тупикових ситуацій з їх завершенням, коли всі вершини неактивні без повідомлень у дорозі. Pregel пропонує C++ API, який є виразним і простим у програмуванні, приховуючи пов’язані деталі розповсюдження. Реалізація Pregel

базується на кластерній архітектурі Google з бібліотекою Pregel, що розбиває вхідний граф на розділи набору вершин та їх ребер, які надалі призначаються робочим машинам. Майстер на місці несе відповідальність за координацію діяльності працівників. Отже, Pregel з його вбудованими функціями представляє себе як спрощену модель для великомасштабних обчислень графів.

Глибоке навчання. Моделі глибокого навчання, що виконують складні завдання, вимагають великих обчислювальних вимог. Ефективне навчання таких моделей вимагає масштабованого підходу, який керується великомасштабними розподіленими системами. Як приклад, Project Adam — це масштабована розподілена навчальна система глибокого навчання, яка компрометує серверні машини, особливо зосереджена на завданнях візуального розпізнавання в глибокій нейронній мережі (DNN). Архітектура системи високого рівня Адама заснована на багатоагентній системі та компромісі з обслуговування даних і машин для навчання моделей і сервера глобальних параметрів, що забезпечує паралельність як моделі, так і даних. Таким чином, це дозволяє розділити велику модель на кілька машин, і, отже, кілька реплік однієї моделі можна навчати паралельно, використовуючи різні розділи набору навчальних даних із загальним набором параметрів, які асинхронно оновлюються ними на глобальному параметрі. сервер. Адам розбиває модель по вертикалі, що мінімізує потребу в міжмашинному зв’язку, а також наявність асинхронних пакетних оновлень параметрів сприяє її масштабованості на кількох машинах. Крім того, багатопотокове навчання моделі досягається на одній машині, дозволяючи потокам оновлювати локальні ваги параметрів без блокувань. Крім того, Адам, як система для навчання великих мереж DNN, демонструє стратегії для пом’якшення впливу дисперсії швидкості через повільні машини, а також містить відмовостійкі стратегії, які можна застосувати на глобальному сервері параметрів.

1.4 Оптимізатори для великомасштабної обробки потоків даних

MapReduce як фреймворк для масивної паралельної обробки має кілька обмежень, коли стосується його використання для ефективної великомасштабної паралельної обробки та аналізу даних. На рис. 1.4 зображено ці області вдосконалення MapReduce.

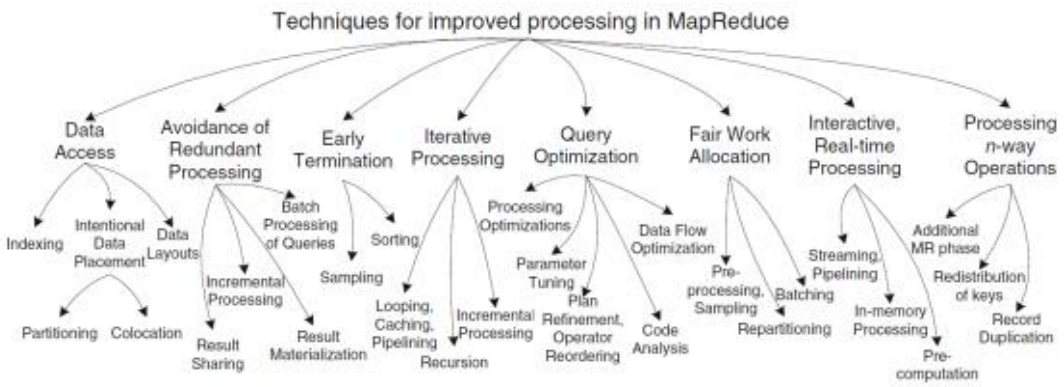

Рисунок 1.4 – Таксономія областей покращення для MapReduce

Враховуючи, що Apache Spark з'явилася де-факто як фреймворк для Big Data Analytics, у цьому розділі нашим основним завданням буде оптимізації, які можна застосувати в усьому стеку Spark, підвищуючи його загальну продуктивність.

На рис. 1.5 зображено архітектурний огляд високого рівня Spark.

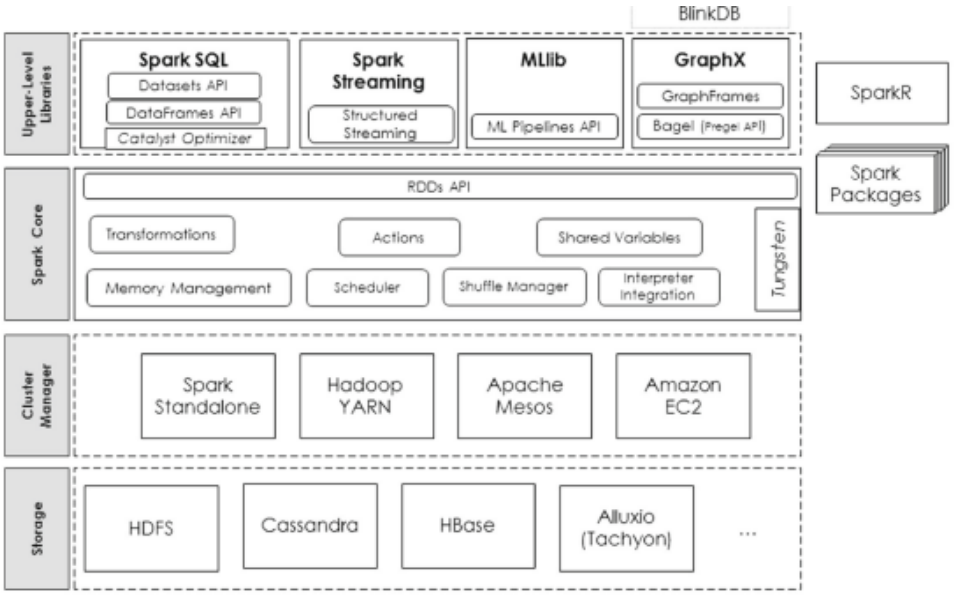

Рисунок 1.5 – Огляд високого рівня стека Apache Spark

Основним компонентом Spark є ядро Spark, написане на Scala і пропонує API на Scala, Java, Python і R для великомасштабної обробки даних. Ядро Spark також пропонує різноманітні функціональні можливості для кластерних обчислень у пам'яті, включаючи перемішування даних, відновлення завдань тощо. Крім того, ядро Spark працює на різних менеджерах кластерів, а саме на Amazon EC2, Hadoop YARN , Apache Mesos та на власному автономному в менеджері кластерів для виконання завдань на різних ресурсах кластера. Крім того, ядро Spark може

отримати доступ до даних з будь-якого джерела даних Hadoop, наприклад. HDFS, Cassandra, HBase, Hive, Alluxio тощо. Різні бібліотеки верхнього рівня створені поверх ядра Spark для роботи з різними робочими навантаженнями, включаючи Spark SQL, що займається обробкою структурованих даних, Spark Streaming для потокової обробки даних, MHlib для масштабування машинне навчання та GraphX, що займаються обробкою графіків. Крім того, різні пакети Spark — це зовнішні пакети з відкритим кодом і бібліотеки, створені для роботи з ядром Spark або бібліотеками верхнього рівня, наприклад. Blink DB як приблизний механізм запитів через Spark SQL. Крім того, SparkR — це внутрішній пакет R, що забезпечує інтерфейс Spark на основі R.

Нижче наведено деякі інструменти, розроблені для оптимізації стека Spark, збільшуючи загальну пропускну здатність.

Проект Вольфрам. Проект Tungsten, розроблений для оптимізації додатків Spark, зосереджений на покращенні ефективності пам’яті та центрального процесору (ЦП) і максимальному використанні базового обладнання. Tungsten як ключовий компонент механізму виконання Spark прагне досягти своїх цілей за допомогою включення трьох основних функцій, а саме, ручного керування пам'яттю шляхом використання семантики програми, яка включає в себе керування пам'яттю програми вручну без необхідності автоматичного керування пам'яттю за допомогою Java Garbage Collector, що мінімізує обсяг пам'яті, зберігання даних у кеші ЦП порівняно з пам'яттю для швидшого доступу до даних, ручне генерування коду, яке передбачає ефективне генерування байт-коду порівняно зі стандартним виконанням Java VM за допомогою сучасних компіляторів і процесорів.

Оптимізатор каталізатора. Оптимізатор каталізатора лежить в основі SparkSQL, що забезпечує ефективний вибір фізичного плану для оптимізації виконання завдань. Catalyst — це розширюваний оптимізатор запитів, створений з використанням функціональних функцій мови програмування Scala, що підтримує оптимізацію як на основі правил, так і на основі витрат. У своїй основі Catalyst складається з бібліотеки, яка виражає вхідні вирази у вигляді дерева, яким можна маніпулювати за допомогою різних правил. Крім того, структура перетворення дерева складається з кількох бібліотек і набору правил, які можна модифікувати користувачем, які оптимізують різні фази виконання запиту, включаючи аналіз і оптимізацію логічного плану, фізичне планування та генерацію коду в байт-код Java. Усі ці фази суто засновані на правилах, за винятком фази фізичного

планування, яка генерує та порівнює декілька планів із виконанням найбільш оптимізованого за витратами. Крім того, будучи розширюваним оптимізатором, Catalyst також пропонує кілька загальнодоступних точок розширення, щоб включати зовнішні джерела даних і визначені користувачем типи, що дозволяє проводити розширену аналітику за допомогою SparkSQL.

RIOS (Інтегрований оптимізатор під час виконання для Spark). RIOS, який можна розглядати як покращення Catalyst, є адаптивним оптимізатором запитів для Spark, який вибирає оптимальний план виконання фізичного запиту під час виконання на основі ідеї залучення пізнього зв'язування багатих і більш релевантних статистичних даних, зібраних під час виконання, без необхідності їх авансовий збір із значними накладними витратами. RIOS спеціально зосереджується на виборі предикатів для даного запиту, щоб визначити оптимальний економічно ефективний порядок об'єднання та реалізацію фізичного з'єднання, і для цього він виконує ітераційну стратегію часу виконання EGAP (Виконати, Збирати, Агрегувати, Планувати), яка використовує лінивий пакет. стратегія для створення та виконання планів запитів у Spark для отримання ефективних планів. Ітеративна стратегія EGAP включає виконання, яке виконує всі етапи даних і попередніх розділів, що відповідають операціям об'єднання в плані фізичного об'єднання, збір, який доповнює етапи Spark для збору більш точної статистики, пов'язаної з атрибутами об'єднання на етапі попереднього розділення, зведення який агрегує зібрану статистику назад до планувальника (Spark Driver), План, який ліниво планує наступний найкращий порядок з'єднання та метод на основі оцінки Наприклад. Кількість приєднання визначається з урахуванням зібраної статистики. Отже, RIOS генерує та модифікує план виконання під час виконання поступово й жадібно, таким чином швидко визначаючи оптимальний план для даного запиту без додаткових витрат, наприклад. Дослідження всього простору плану.

Flare. Flare представляє себе як новий бек-енд для Spark, який покращує його реляційну продуктивність, щоб бути на одному рівні з найкращими двигунами SQL, а також оптимізує його продуктивність за наявності різнорідних робочих навантажень на основі ідеї генерації рідного коду замість коду JVM. Архітектура Flare полегшує його інтеграцію зі Spark на трьох рівнях, Flare Level 1 забезпечує компіляцію власного коду етапів запиту в Tungsten, слугуючи легким прискорювачем для певних запитів, але обмежений в рамках моделі виконання

Spark, рівень Flare 2 забезпечує подальше оптимізація реляційних робочих навантажень за допомогою змін у моделі виконання Spark, що досягає значного прискорення за рахунок компіляції всіх створених Spark планів запитів до рідного коду, усуваючи рівні абстракції Spark під час виконання, Flare Level 3 забезпечує ефективне генерування коду для різнорідних робочих навантажень. завдяки введенню проміжного рівня між планами запитів і кодом, згенерованим у вигляді відображення на високопродуктивну доменно-специфічну мову (DSL), запропоновану фреймворком компілятора Delite.

Apache Calcite. Calcite — це розширювана платформа для обробки й оптимізації запитів, розроблена для роботи з різнорідними робочими навантаженнями для різних типів запитів, серед яких SQL, запити над напівструктурованими, геопросторовими та потоковими даними. Основні архітектурні компоненти Calcite складаються з аналізатора запитів і валідатора, який перетворює запит у дерево реляційних операторів, адаптерів, які забезпечують гнучкість у визначенні основного механізму зберігання, оптимізатора запитів, який використовує реляційне дерево і забезпечує вартість і правила. на основі оптимізації, Relational Builder Interface, який забезпечує вираз вхідних запитів за допомогою реляційних конструкцій. Calcite по суті оптимізує запити у формі реляційних виразів, а також може відображати їх назад у блок обробки запитів конкретної системи. Крім того, Calcite включає в себе підхід динамічного програмування і запобігає потраплянню в локальні мінімуми, одночасно мінімізуючи витрати на виконання запиту в порівнянні з Catalyst, і, отже, може допомогти досягти кращого підвищення продуктивності для Spark SQL.

1.5 Сховище для великомасштабної обробки даних

Розподілена файлова система (DFS) — це файлова система, яка використовується для ефективного зберігання та пошуку даних у розподіленому налаштуванні клієнт-сервер.

Типи розподілених файлових систем (DFS) Більшість розподілених систем вимагають з’єднання з розподіленими файловими системами для досягнення кращої пропускної здатності. Існують різні типи DFS, які задовольняють різні

вимоги розподілених систем. На рис. 1.6 зображені різні типи великомасштабних DFS.

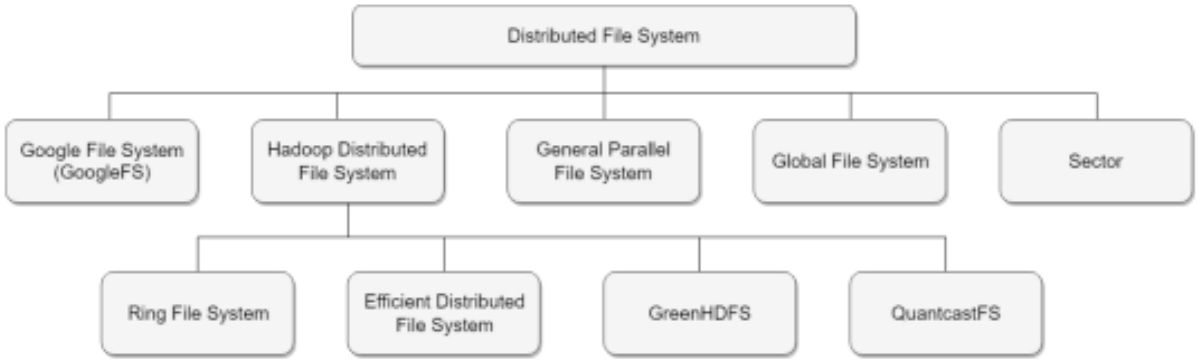

Рисунок 1.6 – Типи розподілених файлових систем

Нижче наведено опис розподілених файлових систем.

Файлова система Google (GoogleFS). GoogleFS широко використовується в Google і відповідно оптимізований на основі вимог Google до зберігання даних і використання. Файли в GoogleFS поділяються на блоки фіксованого розміру по 64 МБ, які супроводжуються 64-бітним дескриптором фрагментів, які потім розподіляються між різними вузлами в кластері, також відомим як сервери фрагментів. Головний вузол несе відповідальність за підтримку метаданих усіх блоків і їх відображення у файлі, а також отримує всі запити від клієнтських вузлів. За запитом головний вузол надсилає ці метадані клієнту, який потім може підключитися безпосередньо до сервера фрагментів для передачі даних. Крім того, GoogleFS може обробляти кілька операцій, що виконуються одночасно на одному фрагменті, за допомогою ефективно розробленого механізму блокування. Крім того, щоб запобігти перевантаженню основного вузла клієнтськими запитами і спричинити його вузьке місце, GoogleFS зберігає метадані в пам’яті головного вузла, що збільшує загальну швидкість обробки. Дані в GoogleFS потрійно реплікуються, щоб використовувати відмовостійкість і відновлення у разі збою вузла.

Розподілена файлова система Hadoop (HDFS). HDFS – це версія GoogleFS з відкритим вихідним кодом, що забезпечує подібну функціональність та

архітектуру і заснована на шаблоні «запис-один раз-читання-багато». HDFS має різні варіанти, зосереджені на різних областях.

Кільцева файлова система (RFS). RFS вважається масштабованою та відмовостійкою DFS, що складається з кількох головних вузлів, які діють як мета-сервери. Ці мета-сервери окремо зберігають частину метаданих за допомогою розподіленої хеш-таблиці (DHT), а також дані потрійно реплікуються на інших мета-серверах. Ця конфігурація полегшує навантаження на один головний вузол і усуває єдину точку відмови.

Ефективна розподілена файлова система (EDFS). EDFS є напівцентралізованим DFS і подібний до RFS в усуненні єдиної точки відмови. На додаток до кількох головних вузлів, він складається з інтерфейсного сервера, який керує сеансами та за допомогою хешування направляє запити на головні вузли. Крім того, EDFS замінює звичайний протокол TCP іншим безіменним протоколом, який є порівняно швидшим, ніж TCP.

GreenHDFS. GreenHDFS фокусується на енергозбереженні у великомасштабному DFS. Вузол даних у GreenHDFS поділяється на дві зони, а саме гарячу зону, яка складається з часто використовуваних або щойно створених файлів, а інші знаходяться в холодній зоні. Крім того, кожен файл пов’язаний з температурою, яка має високе значення для гарячої зони і низьку для холодної зони. В ідеалі підвищення або зниження температури файлу залежить від швидкості доступу, при цьому висока швидкість доступу вказує на підвищення температури і файл знаходиться в гарячій зоні, а низька швидкість доступу знижує температуру файлу, коли файл знаходиться в холодній зона. Існує політика переміщення файлів, яка перевіряє температуру цього файлу та переміщує файл із гарячої зони в холодну і навпаки. Ці холодні сервери переводять у сплячий режим для досягнення оптимальної економії енергії.

QuantcastFS (QFS) QFS зосереджується на забезпеченні кращої продуктивності та економічної обробки у великомасштабній DFS. Він складається з мета-серверів і серверів фрагментів, але в ідеалі один сервер фрагментів на вузол. Крім того, він складається з клієнтського компонента, який діє як інтерфейс, що

представляє API файлової системи іншим рівням програмного забезпечення та полегшує відображення запитів мета-сервера на відповідний сервер фрагментів і безпосередньо взаємодіє з сервером фрагментів.

Загальна паралельна файлова система (GPFS). GPFS – це централізована, паралельна DFS із загальним диском, що демонструє високу масштабованість. Великі файли розділені на блоки, які можна конфігурувати, розміром від 16 КБ до 1 МБ із за замовчуванням 256 КБ. Для зберігання невеликих файлів використовуються підблоки розміром 1/32 звичайного блоку. Крім того, GPFS також забезпечує підтримку великого каталогу, який може містити мільйони файлів з розширюваним хешуванням для пошуку цих файлів. Встановлений розподілений механізм блокування синхронізує доступ до спільних дисків. Що стосується особливих ролей, які відіграють вузли в GPFS, один вузол вибирається в якості диспетчера розподілу, який підтримує статистику вільного простору, доступного в усіх регіонах розподілу, а динамічно обрані метаноди використовуються для централізованого керування метаданими файлів. GPFS також використовує свій власний відмовостійкий механізм, який включає відновлення метаданих і звільнення ресурсів, що зберігаються на збійному вузлі. Він також має політику заміни на випадок, якщо йому потрібно замінити будь-які спеціальні ролі, які виконує несправний вузол.

Глобальна файлова система (GFS). GFS – це DFS на спільному диску, який дозволяє всім вузлам отримати прямий одночасний доступ до спільного блочного сховища. Вузлам у GFS не призначаються спеціальні ролі, і, отже, всі функціонують як однорангові. GFS використовує файлову систему журналування, оскільки кожен вузол кластера має власний журнал, і зміни метаданих файлової системи спочатку записуються в цей журнал, а потім у файлову систему. Це допомагає у разі збоїв вузла відновити узгодженість файлової системи шляхом реплікації цих операцій метаданих, записаних у журналі. Аналогічно, дані також можуть бути записані в журнал, щоб підвищити продуктивність цього відмовостійкого механізму.

Сектор. Сектор може бути розгорнутий на великій території і дозволяє обробляти та обробляти великі набори даних з будь-якого місця. Він складається з головного вузла, сервера безпеки та кількох підпорядкованих вузлів, при цьому сервер безпеки підключений до головного та зберігає інформацію про кількість ведених, інформацію про доступ до файлів тощо. Файли поділяються на секторні фрагменти із зберіганням кожен фрагмент у власній файловій системі підпорядкованого вузла як файл. Щоб забезпечити безпеку даних, підпорядковані пристрої функціонують лише згідно з інструкцією ведучого, а з'єднання між клієнтом і підпорядкованим вузлом встановлюються лише після перевірки ведучим запиту клієнта із сервером безпеки. Дані в Секторі реплікуються потрійно, таким чином забезпечується стійкість до відмов, а вийшли з ладу вузли також можна легко відновити за допомогою необхідних механізмів відновлення. Сектор використовує протокол UDP, який є швидшим у порівнянні зі звичайним протоколом TCP. Крім того, Sector включає надійну бібліотеку, яка називається протоколом групового обміну повідомленнями для передачі повідомлень, що забезпечує додаткову надійність.

База даних, розроблена для великомасштабної обробки даних Bigtable, є прикладом розподіленої системи зберігання, основною метою якої є безпосередня підтримка клієнтських програм, які бажають зберігати і керувати великомасштабними структурованими даними. Bigtable в основному було розроблено як власне рішення Google для зберігання даних і, таким чином, підтримує велику кількість проектів і продуктів Google, включаючи Google Analytics, Google Earth тощо, забезпечуючи високу продуктивність, масштабованість і гнучкість, незважаючи на важкі робочі навантаження. Bigtable використовує просту модель даних, що складається з розрідженої, постійної та розподіленої багатовимірної відсортованої карти, яка індексується за допомогою ключа рядка, ключа стовпця та позначки часу, зі значеннями, які зберігаються у вигляді неінтерпретованого масиву байтів. Діапазони рядків у Bigtable називаються планшетами і утворюють основну одиницю розподілу навантаження та балансування. З іншого боку, ключі стовпців утворюють основну одиницю

контролю доступу шляхом їх групування в набори, які називаються сімействами стовпців. Крім того, часові позначки зазвичай забезпечують контроль версій збережених даних. Ця проста модель даних надає клієнту динамічний контроль над розташуванням і форматом даних, а також дозволяє вказати місцевість даних за допомогою ретельно відібраної схеми даних. Bigtable також надає клієнтський API з різними функціями та функціями, що дозволяє користувачам маніпулювати збереженими даними. Bigtable побудовано з використанням різних будівельних блоків інфраструктури Google, деякі з них включають GoogleFS для зберігання файлів журналів і даних, формат файлу Google SSTable для внутрішнього зберігання даних Bigtable, Chubby як службу розподіленого блокування. Bigtable базується на архітектурі спільного доступу, і його реалізація складається з трьох основних компонентів, а саме: бібліотеки, яка пов’язана з кожним клієнтом для кешування розташування планшета, багатьох планшетних серверів, відповідальних за обробку запитів на читання та запис, керування планшетом і головного сервера. відповідає за функції, включаючи призначення планшета серверам планшетів, керування планшетними серверами. Крім того, Bigtable складається з різноманітних удосконалень у своїй реалізації, включаючи використання фільтрів Блума, механізмів для прискорення відновлення планшетів тощо, що в кінцевому підсумку робить його надійним рішенням для структурованого зберігання даних з високою продуктивністю та доступністю.

2 ОПТИМІЗАЦІЇ ДЛЯ ВБУДОВУВАННЯ ТА МАСИВІВ ДАНИХ

У цьому розділі висвітлюються оптимізації, які можна застосувати для вбудовування як форми даних масиву. Відповідно, пов’язана робота висвітлює два типи оптимізації, а саме на основі моделі вартості та вибору шляху доступу. Крім того, необхідно зупинитися на хешуванні, що зберігає подібність у високих розмірах, при вбудовуванні даних з акцентом на двох підходах, а саме на хешуванні, чутливому до місцевості – Locality Sensitive Hashing (LSH) і навчання хешування – Learning To Hash (L2H).

2.1 Оптимізоване керування вбудовуваннями

У контексті оптимізованого зберігання даних вбудовування, Svensson et al. [20] представляють систематичне дослідження в області архітектури баз даних і точно згадують SciDB як нову ідею наукової бази даних з відкритим вихідним кодом, засновану на моделі даних масиву для підтримки багатовимірних масивів з будь-якою кількістю вимірів. SciDB, спрямований на вирішення проблеми великомасштабного комплексного аналізу наукових даних, підтримує два основних типи операторів над даними масиву, а саме, структурні, що керують структурою масиву незалежно від даних, наприклад, переформатування та залежно від вмісту, що виконує залежний від даних вміст. операції на основі, наприклад, агрегація. Крім того, завантаження даних у SciDB включає масове завантаження даних зі сховищем, організованим як відображення даних у сегменти диска на прямокутні фрагменти масиву. Структура даних R-дерева відстежує розташування дисків і вміст відро. Крім того, оптимізації (обговорюються як відкриті дослідницькі області в цій статті) можуть бути застосовані до цього макета сховища, щоб визначити оптимальний розмір ковша, стиснення ковша, об’єднання ковшів тощо, що також призведе до підвищення продуктивності. Крім того, як інший варіант зберігання масивів, TileDB є багатовимірним менеджером зберігання масивів, оптимізованим як для щільних, так і для розріджених масивів. Його основна ідея полягає в тому, що дані елементів масиву організовані в упорядковані колекції, відомі як фрагменти, які можуть бути щільними або розрідженими, і групувати безперервні елементи масиву в плитки даних фіксованої

ємності. TileDB як новий менеджер сховища, отже, підтримує розпаралелювання, пропонуючи швидше читання та запис.

Крім того, в області оптимізованої обробки запитів горизонталей для багатовимірних даних представлений ефективний, масштабований і надійний алгоритм паралельного обчислення горизонталей, реалізований на платформі Hadoop MapReduce. Цей алгоритм спеціально вирішує дві важливі проблеми, а саме: спотворення даних та відставання даних, з якими стикаються при обробці запитів горизонталей у розподілених середовищах із дуже великими наборами даних.

Крім того, як ще один приклад багатовимірних векторних даних, великі просторові векторні дані (BSVD) включають широкий спектр типів просторових даних, наприклад, картографічні дані, дані на основі розташування і т. д., і можуть бути представлені точками, лініями і або площі полігонів. Yao та ін [21] представляють всебічний огляд сучасних технологій та методів, пов'язаних з управлінням даними та обробкою BSVD.

Що стосується оптимізованих схем зберігання, гібридні схеми стають все більш популярними для оптимізації компонування для виконання спеціального аналізу даних при великомасштабній обробці даних. Гібридний макет зазвичай ділить необроблені дані на горизонтальні розділи, де дані зберігаються вертикально всередині кожного розділу за допомогою кодування та стиснення даних. Однак для цих макетів потрібні параметри, що настроюються, наприклад розмір горизонтального розділу, який може безпосередньо впливати на продуктивність запиту. Отже, ATUN-HL, створений для Apache Parquet (реалізація гібридного компонування з відкритим вихідним кодом HDFS), забезпечує автоматичне налаштування цих параметрів для визначення їх найкращого можливого значення на основі оптимізації на основі витрат, заданої в якості вхідних даних і робочої навантаження. Характеристики.

У контексті оптимізованої обробки запитів за допомогою Apache Spark до даних, закодованих Apache Parquet, Braams et al. [23] експериментально вивчають метод оптимізації Predicate Pushdown, який прискорює вибіркові запити за рахунок передачі операцій фільтрації оператор сканування, який відповідає за читання даних, і в дослідженні також представлена оптимізована дрібнозерниста реалізація Predicate Pushdown в контексті цього каркас обробки даних. Отже, це дослідження з різними сценаріями додатків демонструє значне збільшення продуктивності

запитів за рахунок використання можливостей ранньої фільтрації для запитів, що включають предикати рівності і діапазону або вибіркові з'єднання. Цей метод може бути розумно застосований до вбудованих даних з пропуском на основі схожості з використанням хеш-кодів, що зберігають схожість.

2.2 Багатомірне хешування

В останні роки пошук за методом наближеного найближчого сусіда (ANN) став помітною темою досліджень для ефективної обробки обсягу даних, що постійно зростає, в реальних додатках. Серед існуючих методів ІНС хешування стало популярним завдяки високій швидкості запитів та низьким витратам пам'яті. Модель хешування бере точку вхідних даних (зображення, документ тощо) і виводить послідовність бітів або хеш-код, що представляє цю точку даних. Таксономія моделей хешування, що ґрунтується на їх фундаментальних властивостях, як показано на рис. 2.1, в цілому ділить їх на дві категорії: проекція, яка фокусується на вивченні низькорозмірного перетворення вхідних даних таким чином, щоб пов'язані точки даних розташовувалися ближче один до одного в моделі хешування. простір та квантування, які зосереджені на перетворенні цих проекцій у двійкові біти з використанням порогового механізму.

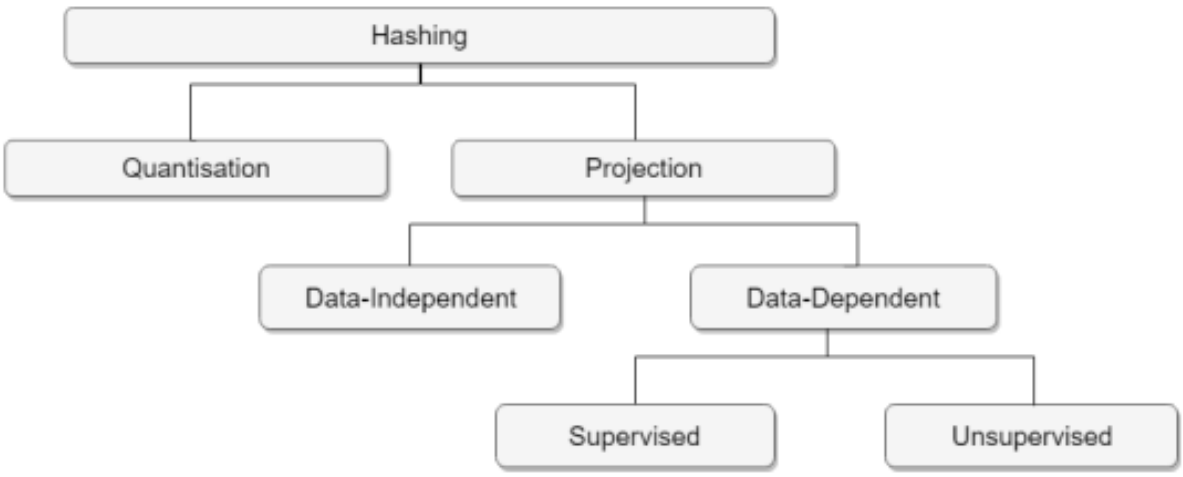

Рисунок 2.1 – Таксономія моделей хешування

Проекція може бути розділена на дві категорії: незалежна від даних, коли хеш-функція вивчається випадковим чином незалежно від розподілу вхідних даних, а репрезентативні методи включають хешування з урахуванням

розташування (LSH) і його варіанти, і залежні від даних , які враховують облік розподілу вхідних даних та репрезентативні методи включають методи Learning To Hash (L2H). Залежні від даних моделі можна далі розділити на контрольовані, які вивчають хеш-функцію, використовуючи контрольовану інформацію у вигляді міток, щоб максимізувати появу пов'язаних точок даних, які повинні бути хешовані в одні й ті ж сегменти, і неконтрольовані, які вивчають хеш-функцію, не вимагаючи контрольованого. маркувати інформацію зазвичай за допомогою факторизації даних. У порівнянні з залежними від даних методи, незалежні від даних, можуть забезпечити порівнянну або навіть кращу точність з більш короткими хеш-кодами і, отже, стають все більш популярними у реальних додатках. Отже, ці методи багатовимірного хешування можна розумно вивчити щодо їх використання ефективного управління та вилучення вбудованих даних.

2.3 Хешування, чутливе до місцевості

Locality Sensitive Hashing (LSH) - це наближений незалежний від даних масштабований алгоритм, який забезпечує швидше порівняння між точками в дуже багатовимірному просторі за допомогою їх випадкової проекції на низькорозмірні бітові сигнатури, приблизно зберігаючи їх косинусну відстань. LSH хешує точки вхідних даних у кошики таким чином, що схожі точки даних зіставляються з одним і тим самим кошиком з високою ймовірністю, а точки даних, віддалені один від одного, швидше за все, поміщаються в різні кошики. Отже, LSH знаходить своє застосування в кількох областях, деякі з яких включають виявлення майже дублікатів, ієрархічну кластеризацію, повногеномне дослідження асоціацій і т. д. Існують різні види LSH і один з них - бінарний. LSH, який генерує двійкові коди для точок вхідних даних та обчислює приблизну відстань або подібність між ними через відстань Хеммінгу для цих двійкових кодів, використовуючи в основному побітові операції. Це діє як масштабоване рішення у великомасштабних додатках, оскільки ці двійкові обчислення набагато швидші та ефективніші для зберігання.

2.4 Вивчення хешу

Learning To Hash (L2H) – це набір методів хешування, що залежать від даних, які спрямовані на вивчення компактного побітового уявлення з більш короткими

хеш-кодами таким чином, щоб аналогічні вхідні дані відображалися в сусідні двійкові хеш-коди. Методи L2H поділяються на дві категорії, а саме: унімодальні та мультимодальні, кожен з яких складається з методів навчання, таких як контрольовані, які використовують контрольовану інформацію про мітки для вивчення хеш-кодів під час процедури навчання, та неконтрольовані, які використовують тільки інформацію про характеристик або атрибутів точок даних без будь-якого контрольованого маркувати інформацію під час процедури навчання. Можливість надання інформації про контрольовані мітки досягається в трьох різних формах, а саме: точкові мітки, парні мітки та ранжируючі мітки. Деякі з унімодальних методів навчання з учителем включають напівкероване хешування (SSH), хешування з мінімальними втратами (MLH), хешування на основі лінійного дискримінантного аналізу (LDAHash), моделі прихованих факторів для контрольованого хешування (LFH, FastH, контрольоване дискретне хешування (SDH), дискретне контрольоване хешування на основі вибірки стовпців (COSDISH). Крім того, деякі з унімодальних методів навчання без вчителя включають аналіз основних компонентів (PCAH), спектральне хешування (SH), хешування (STH), хешування на основі графа прив'язки (AGH) .], ітеративне квантування (ITQ), IsoHash, хешування дискретних графів (DGH), хешування масштабованих графів (SGH). Мультимодальне хешування являє собою хешування з декількома джерелами, яке спрямоване на вивчення якісніших двійкових кодів, ніж одномодальне хешування, за рахунок використання допоміжних уявлень, що передбачають, що всі уявлення надаються для запиту, і включає методи, а саме, хешування множинних ознак (MFH), складове хешування і крос-модальне хешування, яке повертає елементи, аналогічні елементам вхідного запиту, і включає такі методи, як хешування перехресного представлення (CVH), мультимодальне приховане двійкове вбудовування (MLBE), Ко-Регуляризоване хешування (CRH), міжмедійне хешування (IMH), гетерогенне хешування з урахуванням відносин (RaHH), максимізація семантичної кореляції (SCM), х матричної факторизації (CMFH), квантоване кореляційне хешування (QCH), хешування із збереженням сем антики (SePH). Хешування на основі ранжирування, що являє собою L2H-компроміс методів, заснованих на ранжируванні, включаючи дистанційне метричне навчання Хеммінга (HDML), хешування із збереженням порядку (OPH), контрольоване хешування на основі ранжування (RSH), хешування зі збереженням ранжирування (RPH), які підтримуються контрольованою

інформацією у вигляді ранжирующих міток для даних. Крім того, глибоке хешування, що становить контрольований L2H, використовує методи глибокого навчання, компрометуючі методи хешування, включаючи хешування згорткової нейронної мережі (CNNH), хешування мережі в мережі (NINH), хешування на основі глибокого семантичного ранжування, глибоке Хешування з регуляризованим порівнянням подібності (DRSCH), глибоке хешування (DH), глибоке парне контрольоване хешування (DPSH).

Глубокое хеширование.

Приймаючи увагу методи глибокого хешування з літератури, які будуть використовуватися при розробці нашого методу L2H, Li та ін. [16] запропонували новий метод глибокого хешування під назвою Deep Pairwise-Supervised Hashing (DPSH), який одночасно включає навчання та вивчення хеш-кода з використанням контрольованих парних методів у складній архітектурі, користуючись механізмом зворотної зв'язку для кращого вивчення хеш-кодів. Ця наскрізна структура навчання, показана на рис. 2.3, для пошуку зображень складається з трьох ключових компонентів: перший компонент — це сверточна нейронна установка (CNN) для навчання представлення зображень із пікселів, другий компонент — хеш-функція, яка відображає вивчену інформацію. представлені зображення в хеш-коді, а третій компонент представляє собою функцію втрати, яка вимірює згенерований хеш-код відповідно до парними мітками.

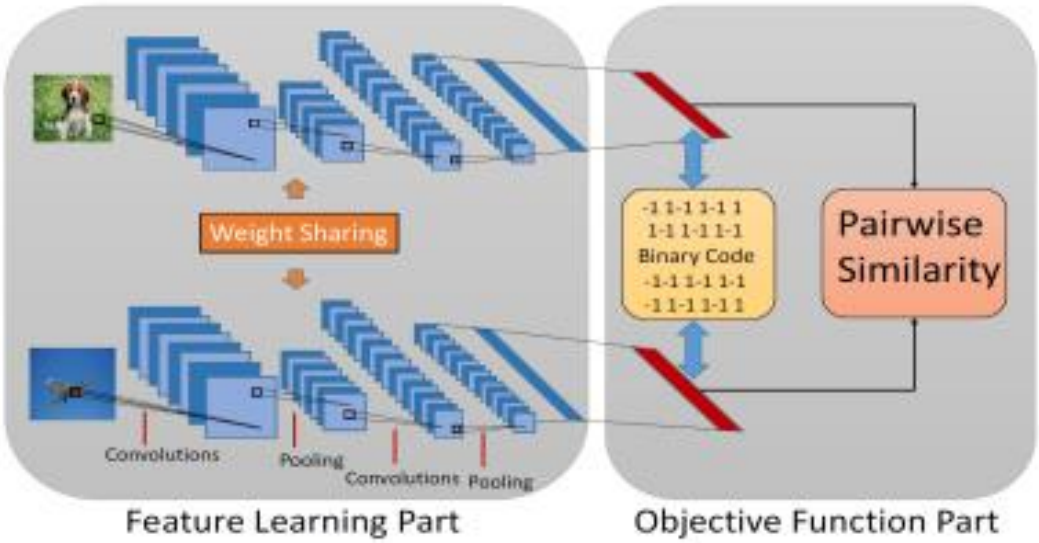

Рисунок 2.3 – Наскрізна архітектура глибокого хешування з попарним контролем (DPSH)

Натхненні цією роботою, Wang та ін. [16] пропонують новий метод глибокого хешування на основі триплетів (частий випадок ранжування міток), який одночасно виконує наскрізне вивчення визнаних зображень і хеш-коду, прагне максимізувати ймовірність вхідних триплетних міток. Експериментально оцінюється, що цей метод перевершує глибоке хешування на основі парних методів і існуючих методів глибокого хешування на основі триплетних методів. Ця наскрізна структура навчання, як показано на рис. 2.4, складається з трьох компонентів: глибокої нейронної мережі для вивчення функцій зображень, одного повністю зв'язаного слою для вивчення хеш-кодів для цих функцій і функцій втрат.

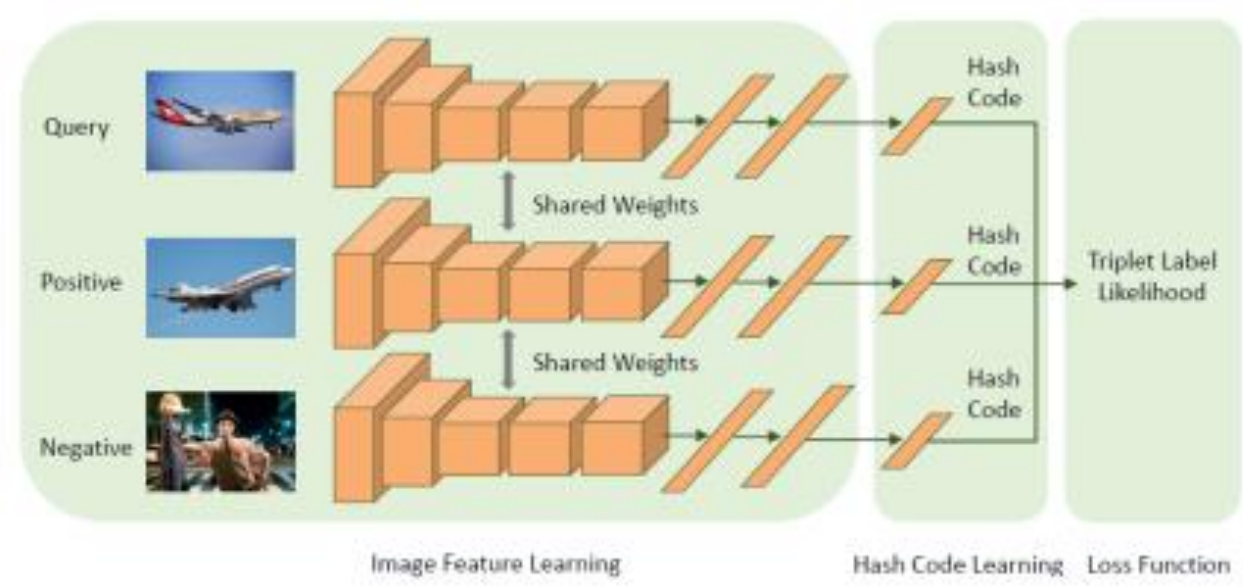

Рисунок 2.4 – Архітектура глибокого хешування на основі триплетів

Чжан та ін. [17] був першим, хто запропонував метод контрольованого хешування, латентний фактор хешування (LFH), для вивчення збереження збігу подвійних кодів на основі моделей скритих факторів, які можна ефективно використовувати для навчання крупномасштабних задач контрольованого хешування. LFH моделює ймовірність попарного збігу як функцію розставання Хеммінга між відповідними точками. Однак оптимізація цієї цільної функції являє собою проблему дискретної оптимізації, яку нелегко вирішити, і, следовательно, це обмеження послаблюється шляхом перетворення дискретних кодів у неперервні, які не можуть забезпечити задовільну продуктивність. Наступні, методи глибокого хешування, які використовують LFH, пропонують нові стратегії для вирішення цієї задачі дискретної оптимізації дискретним способом, що підвищує точність. Wang

та ін. [16] скоріше вирішують цю проблему дискретної оптимізації неперервним способом, як це було запропоновано Zhang et al. [17], але враховує помилку квантування, викликану цю релаксацію у функції потері. Крім того, щодо оптимізованої функції потері для попарного контролюючого хешування Zhu et al. [18] пропонує нову архітектуру мережі глибокого хешування (DHN), яка може одночасно оптимізувати втрати ентропії попарно і перехрестно для вивчення семантично схожих пар і втрат при парному квантуванні для вивчення хеш-кодів. Це служить перевагою для кращого навчання із збереженням подібності та контролю якості вивчених хеш-кодів. Конвеєрна архітектура DHN, показана на рис. 2.5, складається з чотирьох ключових компонентів: перший компонент являє собою підмережу, сучасну з кількох слоїв об'єднання сверток для вивчення представлених зображень, другий компонент являє собою повнозв'язний рівень хешування для генерації компактних хеш-кодів. третій компонент являє собою функцію потері парної перехрестної ентропії, а четвертий компонент являє собою функцію потері парного квантування.

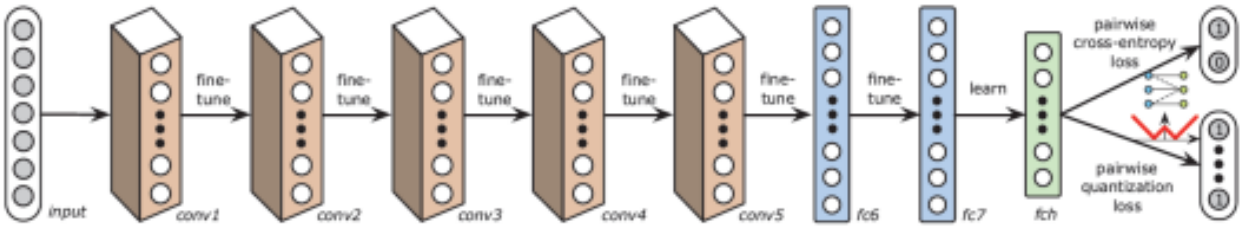

Рисунок 2.5 – Архітектура мережі глибокого хешування

Li і др. [19] пропонують новий метод глибокого хешування Deep Supervised Discrete Hashing (DSDN), який використовує як попарні мітки, таку й класифікаційну інформацію для генерації дискретних хеш-кодів у рамках одного потоку. Процес оптимізації зберігає цей дискретний характер хеш-кодів, щоб зменшити помилку квантування, і, таким чином, метод альтернативної мінімізації виводиться для оптимізації функцій втрат.

2.5 Постанова підконтрольної організації

Entity Resolution (ER) – це процес ідентифікації записів в інформаційних системах, які відповідають одній і тій же реальній сутності. На рис. 2.6 зображено загальний перебіг процесу для ER.

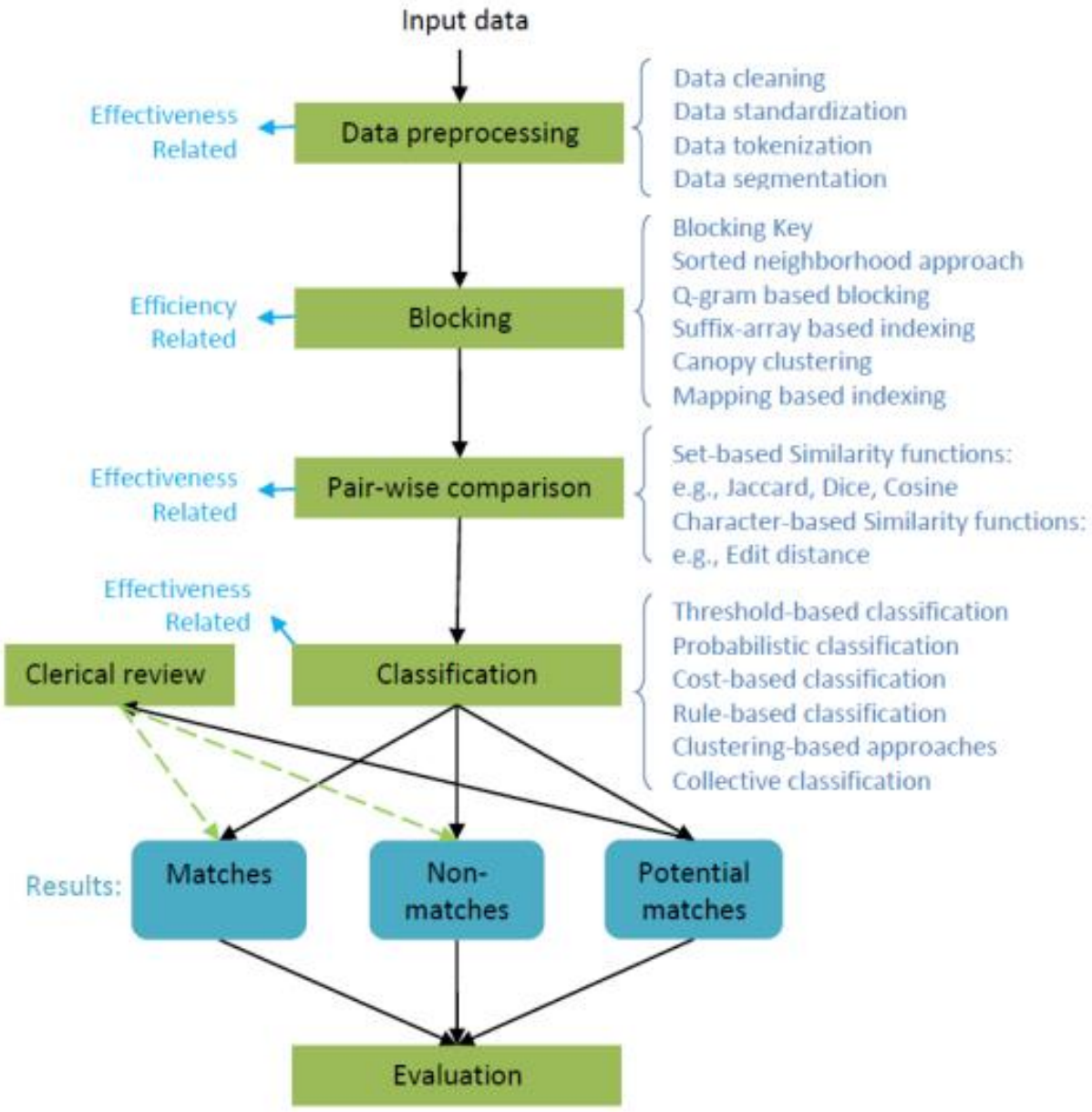

Рисунок 2.6 – Загальний процес розгляду суб'єктів господарювання

3 ОГЛЯД ПРОЕКТУ ТА РЕАЛІЗАЦІЯ ПРОТОТИПУ

3.1 Огляд проекту

У цьому розділі представлений огляд високого рівня дослідницької області, а потім він зіставляється з фактичною темою цієї дипломної роботи. На рис. 3.1 зображено цей огляд високого рівня.

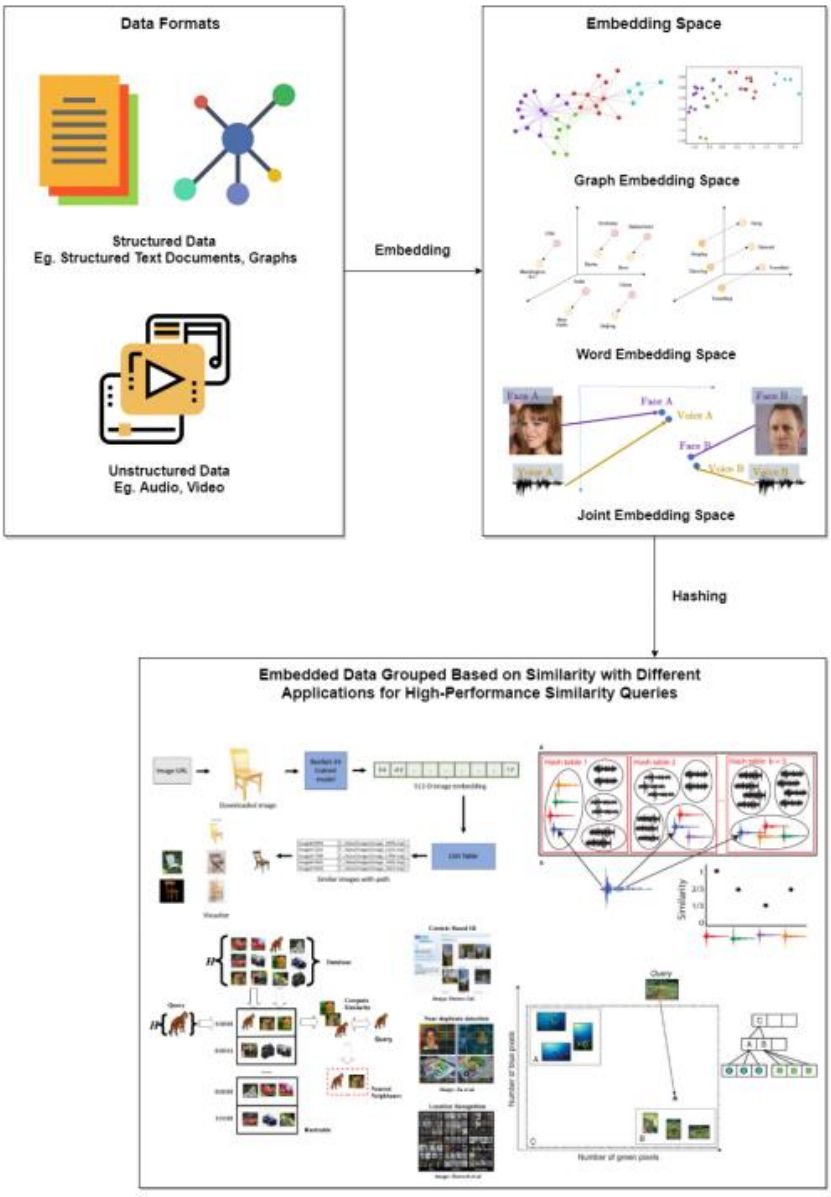

Рисунок 3.1 – Огляд високого рівня досліджуваної області

У дослідницькій сфері ефективного керування вбудованими даними для використання в програмах машинного навчання (ML) у нас є різні формати вхідних даних. Ці формати даних можуть бути структурованими (наприклад, структуровані текстові документи, графіки) або неструктурованими (наприклад, аудіо, відео). Використовуючи методи вбудовування цих форматів даних, такі як методи, засновані на підрахунку (наприклад, тематичні моделі) або на основі передбачення (наприклад, нейронні моделі) можна отримати їх векторне представлення в низьковимірному вбудовуванні space, що тепер покращує керованість даними, висвітлюючи семантичну подібність даних у єдиному просторі. Як приклади, на рис. 3.1 можна різні простори вбудовування для графіків, слів і об'єднаних/комбінованих випадків (зображення та аудіо). Крім того, для ефективного зберігання та керування цим вбудовуванням можна використовувати різні методи хешування, що зберігають подібність, такі як Locality Sensitive Hashing (LSH) або Learning To Hash (L2H) [23]. Це сприяє підвищенню швидкості запитів і зменшенню витрат на пам'ять за рахунок хешування подібних даних разом із високою ймовірністю. Таке блокування вбудованих даних у групи даних на основі їхньої схожості може сприяти вирішенню різних завдань ML, навіть не тільки для ефективності управління даними. Як приклади, на рис. 3.1 можна візуалізувати ефективність завдань пошуку зображень із найближчими сусідами, пошук інформації на основі вмісту, виявлення майже дублікатів, розпізнавання місцеположення [22], отримання форми сигналу для виявлення землетрусу.

3.2 Пропонований підхід

Для вирішення питань дослідження можна використовувати існуючі найсучасніші реалізації для незалежних від даних LSH (наприклад, java-LSH). Зокрема, можна обрати Super-Bit (SBLSH) через його простоту та здатність надавати неупереджені оцінки кутової подібності.

Було розроблене нове рішення для контрольованого хешування на основі нейронних мереж. Цей метод спирається на нейронну мережу глибокого хешування (DHNN), як показано на рис. 3.2.

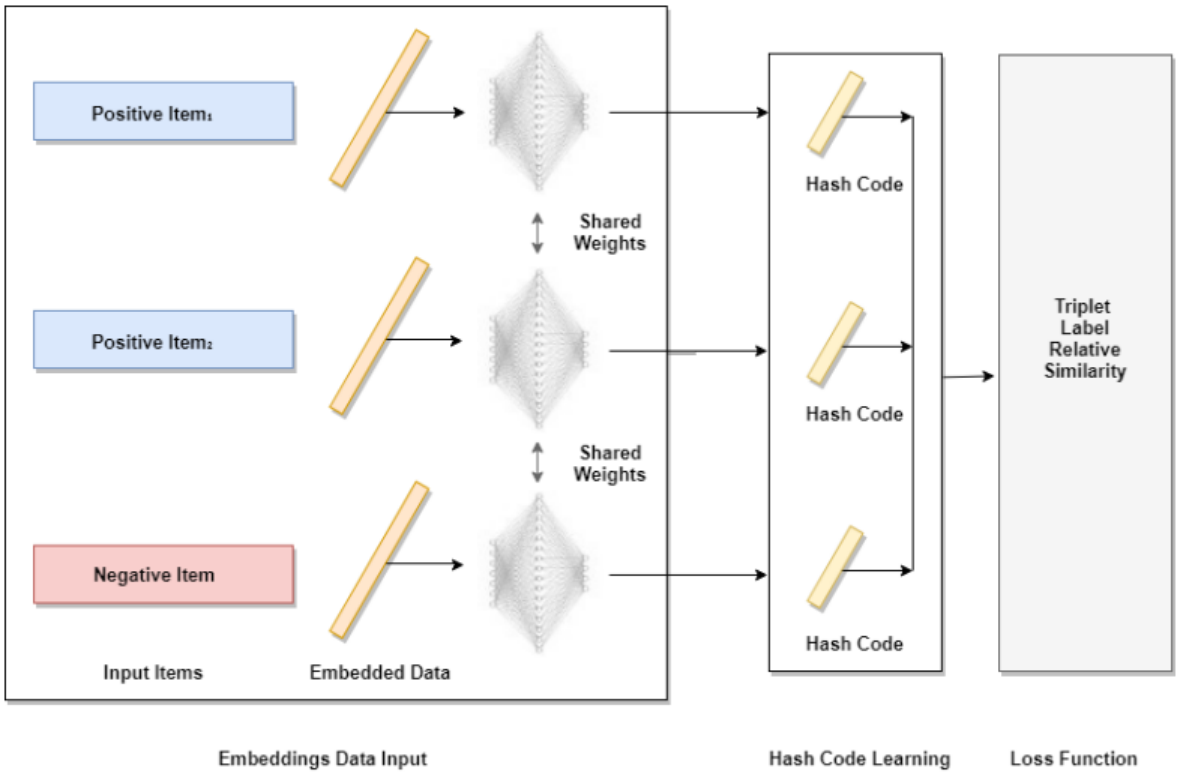

Рисунок 3.2 – Огляд запропонованого методу глибокого хешування для нейронної мережі глибокого хешування (DHNN)

На це вплинула робота Wang та ін.[16], яка є прикладом дослідження глибокого навчання під наглядом на основі триплетних міток. Триплетні мітки – це особлива форма ранжувальних міток для навчальних цілей, які містять більш багату інформацію, ніж попарні мітки, і які можна природним чином розкласти на три попарні мітки. Крім того, використання триплетних міток у вивченому просторі хеш-коду полегшує вивчення кодів, які зберігають близькість позитивних вхідних даних і далеку відстань для негативних вхідних даних, допомагаючи уявленню про відносну схожість між входами.

3.3 Визначення проблеми

За допомогою двох вбудованих наборів даних, кожен із яких складається з вбудовування вхідних слів, пов’язаних з даною сутністю, ідентифікованою ключем (ID), і основну істину подібності, що складається з пар ключів (ідентифікаторів) для подібних сутностей, які діють як постачальник контрольованої мітки інформації. Приклади, позначені трійкою для контрольованого навчання з

використанням цих даних, можна визначити як: $T = \{(P1_0, P2_0, N_0),....,(P1_n, P2_n, N_n)\}$ де n позначає розмір навчальної партії, а в знаходяться навчальні приклади, кожен з яких утворений випадковим вибором двох подібних елементів (по одному для кожного набору даних) як P1 і P2 і одного різного елемента як N (з будь-якого набору даних), на основі позначеного набору даних про основну істину. На основі цих позначених триплетом прикладів відповідні вбудовані дані передаються як вхідні дані до мережі у вигляді трьох масивів, перші два масиви яких містять подібні елементи, а третій масив містить різні елементи. Ці масиви заповнюються з позицій від 0 до n = розмір навчальної групи, утворюючи групу точок. Елементи у відповідних позиціях трьох масивів відповідають трійці для навчальних цілей.

3.4 Вивчення хеш-функції

Для цього завдання був створений прототип нейронної мережі глибокого хешування (DHNN), щоб вивчити хеш-функцію. Огляд методу представлений на рис. 3.2. Він складається з трьох ключових компонентів, а саме вбудованого введення даних, вивчення хеш-коду та функції втрати.

Вбудований вхід даних Цей компонент складається з попередньо вбудованих векторів, що надаються як вхідні дані, що відповідають прикладам із міченими триплетами, сформованими на основі базового набору даних істини. Таким чином, навчальний вхід у вигляді двох позитивних (подібні елементи) і одного негативного (різнорідні елементи) масивів, заповнених від 0 до n = розмір навчального пакету і подається в мережу (для навчання) по одній партії прикладів за раз. Функція навчання, що покращує продуктивність хешування, може бути додатково включена як наскрізний метод глибокого хешування, щоб трохи покращити вбудовування під час навчання.

Вивчення хеш-коду. Цей компонент призначений для вивчення хеш-кодів, що відповідають вхідним даним вбудовування. Зокрема, було використано повністю підключену нейронну архітектуру з вхідним шаром, таким же великим, як передбачуваний вхід, одним прихованим шаром із загальною кількістю 5120 нейронів і вихідним шаром із розміром цільового хеш-коду. Було застосовано функцію активації Rectified Linear Unit (ReLU) у прихованому шарі, але у вихідному шарі активація не застосовується, та було обрано значення діапазону

(-1, +1). Після завершення навчання для створення фактичних хеш-кодів використовується функцію активації знаку, щоб відобразити їх на 0 і 1.

Функція втрати Цей компонент є ключовим компонентом для процедури навчання. Він оцінює ймовірність вивчених хеш-кодів до вірогідності вхідних елементів, позначених триплетом. Метою цієї функції втрат є направляти оптимізацію ваг нейронної мережі, щоб у вхідному прикладі вона призводила до хеш-коду, який мінімізує відстань до хеш-коду аналогічного вхідного прикладу та збільшує відстань до різнорідного прикладу. приклад. З цією метою навчання відбувається групово, причому кожен пакет складається з трійних прикладів. Як перший крок, хеш-коди для кожного елемента обчислюються за допомогою нейронної мережі в її поточному стані.

На основі ймовірності парної подібності через хешування латентного фактору (LFH) і під впливом роботи Li et al. та Wang et al., визначається функція втрат наступним чином:

Дано два хеш-коди b_i та $b_j \in \{-1,+1\}^L$ з набору хеш-кодів B і L як наданої довжини хеш-коду, Θ_{ij} позначає половину внутрішнього добутку між ними,

$$\theta_{ij}=\frac{1}{2}\cdot b_i^T \cdot b_j. \tag{3.1}$$

Ймовірність парних міток $S = \{s_{ij}\}$ можна визначити як,

$$p\left(s_{ij}|B\right)=\begin{cases} \sigma\left(\theta_{ij}\right), s_{ij}=1 \\ 1-\sigma\left(\theta_{ij}\right), s_{ij}=0 \end{cases}, \tag{3.2}$$

де $\sigma(\theta ij)=\frac{1}{1+e^{-\theta_{ij}}}$ – сигмовидна функція і $s_{ij} = 1$ для схожих міток і $s_{ij} = 0$ для різнорідних міток.

Крім того, функція втрат може бути визначена як негативний журнал цієї ймовірності парних міток, яка обчислюється за формулою:

$$\alpha=-\log p(S|B)=-\sum_{s_{ij}\in S}\log p(s_{ij}|B)=-\sum_{s_{ij}\in S}\left(s_{ij}\cdot\theta_{ij}-\log\left(1+e^{\theta_{ij}}\right)\right). \quad (3.3)$$

Формула (3.3) для парних міток модифікована, щоб включити триплетні мітки $T = \{(P1_0, P2_0, N_0),....,(P1_n, P2_n, N_n)\}$. Це представлено в розгорнутому вигляді наступним чином:

$$\alpha=-\sum_{s_{ij}\in S}\begin{pmatrix} s_{p1_np2_n}\cdot\theta_{p1_np2_n}+s_{p1_nN_n}\cdot\theta_{p1_nN}+s_{p2_nN_n}\cdot\theta_{p2_nN_n} \\ -\log\left(1+e^{\theta_{p1_np2_n}}\right)-\log\left(1+e^{\theta_{p1_nN_n}}\right)-\log\left(1+e^{\theta_{p2_nN_n}}\right)\end{pmatrix}. \quad (3.4)$$

Оскільки P1, P2 є позитивними подібними мітками, отримуємо $s_{P1nP2n} = 1$ та P1, N та P2, N як різнорідні мітки, та отримуємо $s_{P1nNn} = 0$ та $s_{P2nNn} = 0$. Підставляючи ці значення в формулу (3.4), отримується функцію втрат, яка буде використовуватися як

$$\alpha=-\sum_{s_{ij}\in S}\left(\theta_{p1_np2_n}-\log\left(1+e^{\theta_{p1_np2_n}}\right)-\log\left(1+e^{\theta_{p1_nN_n}}\right)-\log\left(1+e^{\theta_{p2_nN_n}}\right)\right). \quad (3.5)$$

Мінімізація формули (3.5) є важкодоступною для вирішення проблеми дискретної оптимізації, тому вона вирішується шляхом ослаблення двійкових кодів $\{b_n\}$ від дискретних до безперервних дійсних векторів $\{u_n\}$, де$\{u_n\} \in R^{L\times 1}$ як запропоновано в роботі Zhang et al. [17]. Тому тепер Θ_{ij} можна повторно визначити як:

$$\theta_{ij}=\frac{1}{2}\cdot u_i^T\cdot u_j. \quad (3.6)$$

Але цей процес викликає помилку релаксації, що призводить до зниження продуктивності. Це також відомо як помилка квантування, викликана в процесі квантування вивчених оптимальних дійсних векторів $\{u_n\}$ до двійкових кодів $\{b_n\}$. Щоб зменшити цей вплив на продуктивність, пропонується збалансувати рівняння,

помноживши доданок Θ_{P1mP2m} три рази (оскільки розглядалися триплети) на функцію втрат і, отже, вивести кінцеву функцію втрат для мінімізації як:

$$\alpha=-\sum_{s_{ij}}\left(\begin{array}{l} 3\cdot\theta_{p1_n p2_n}-\log\left(1+e^{\theta_{p1_n p2_n}}\right)- \\ -\log\left(1+e^{\theta_{p1_n N_n}}\right)-\log\left(1+e^{\theta_{p2_n N_n}}\right) \end{array}\right). \quad (3.7)$$

Слід зазначити, що додавання цього множення, а не кількісної помилки виявилося рішенням, яке добре працювало емпірично в наших експериментах. Незважаючи на це, можна також розглянути альтернативні рішення.

3.5 Модельне навчання

Нейронна мережа глибокого хешування (DHNN), як показано на рис. 3.3, є повністю підключеною глибокою нейронною мережею з трьома шарами: вхідним, прихованим і вихідним.

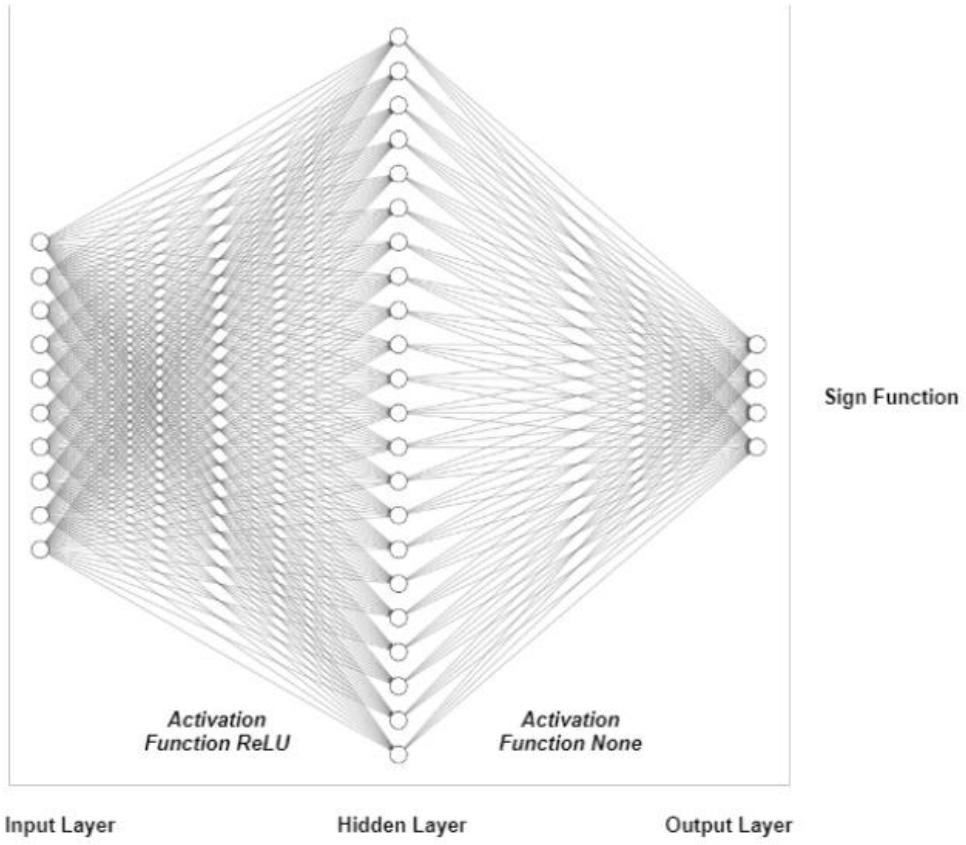

Рисунок 3.3 – Нейронна мережа глибокого хешування

Розмір вхідного шару відповідає розміру вбудованих елементів, тобто 300 у випадку хешування однієї функції та 901 у випадку хешування всіх функцій. Прихований шар являє собою повністю підключений шар із 5120 нейронів і вивчає оптимальний хеш-код, відповідний елементу введення. Крім того, між цими двома шарами використовується функція активації Rectied Linear Unit (ReLU), яка демонструє перевагу простого обчислення та меншої кількості зникаючих градієнтів, що призводить до кращого навчання. Нарешті, вихідний рівень виводить хеш-код після мінімізації функції втрат (рівняння 3.7). Розмір вихідного шару відповідає вказаній довжині хеш-коду, тобто 12, 16, 24, 32, 48, 64, 128. Отже, під час навчання мережі він приймає як вхідний масив вбудованих елементів (два позитивних і один негативний), заповнений від 0 до n = розмір пакету навчання один за одним і обробляючи елемент за елементом для виведення відповідних масивів хеш-кодів (для два позитивних і один негативний) при мінімізації втрат формули (3.7) за допомогою алгоритму зворотного поширення.

На кожній ітерації обирається 50 пакетів і оптимізуємо для двох епох. Після завершення навчання отримуються хеш-коди, що відповідають кожному з вбудованих вхідних даних, зіставлені з його ключем (ID), а потім застосовується до цих хеш-кодів функцію sign, яка перетворює їх у формат 1s і 0s.

Під час оцінки створюється детальний звіт із покриттям метрик, обчисленням та варіаціями покриття з сусідніми збігами хешів у кінці кожної ітерації. Тому за тренінг створюється 2000 звітів. Для експериментальної оцінки враховуються останні повідомлені значення. Навчання збігається приблизно на 1000-й ітерації. Після останньої ітерації також створюється звіт із зазначенням значень покриття та обчислень, досягнутих за допомогою підходу грубої сили (Brute Force), який виступає в якості базової лінії для оцінки продуктивності навченої мережі. Підхід грубої сили зазвичай вирішує певну проблему шляхом вичерпного перерахування всіх можливостей, враховуючи всі можливі результати для прийняття рішення. Було розглянуто підхід грубої сили для виконання хешування таким чином, щоб він генерував максимальне покриття та мінімальне значення обчислень. При цьому він вичерпно намагається призначити подібним елементам подібні значення хеш-коду, максимізуючи таким чином охоплення та мінімізуючи обчислення. Крім того, для деяких випадків також оцінюється продуктивність навченої мережі за допомогою базового підходу грубої сили.

3.6 Попередня обробка та векторизація вхідних наборів даних

Для попередньої обробки та векторизації даних ці пари наборів даних були додатково вбудовані лише для атрибутів String, за винятком поля ID, за допомогою попередньо навченої нейронної моделі fastText Facebook, при цьому кожне вбудовування має 300 вимірів і загальну довжину вхідного вектора. 901. Вставляється кожне слово в атрибути окремо, а як вбудований вектор включаємо суму цих вкладень, поділену на кількість слів у атрибуті. Робиться це для кожного атрибута рядка в оцінюваних наборах даних. Останній запис у цьому векторі відповідає нормалізованому значенню (ціна або рік, відповідно до набору даних). Для розрахунку подібності між парами, що підтримує задачу розв'язання контрольованого об'єкта, обчислюється косинусна подібність між векторами кожного атрибута, а для останньої ознаки в наборі даних використовуємо абсолютну різницю над нормованими значеннями. Як результат, об'єктами, які використовуються для завдання вирішення контрольованого об'єкта, є три подібності косинуса (для вбудованих об'єктів) і одна абсолютна відмінність (для невбудованих об'єктів).

3.7 Метрики оцінювання

Було визначено декілька важливих показників оцінки, які використовуються відповідно до нашої експериментальної оцінки.

Покриття. Визначена користувачем метрика, яка оцінює, скільки співставлень ключів у наборі даних Ground Truth фактично збігається з відображеннями ключів, здійсненими за допомогою вибраної техніки хешування.

$$\text{Coverage}=\left(\frac{100\cdot \text{Number of Hash Matches}}{\text{Number of True Matches}}\right). \tag{3.8}$$

Обчислення Визначена користувачем метрика, яка оцінює, як сума декартового добутку ключів у кожному хешованому сегменті відноситься до загального декартового добутку, таким чином він фіксує розподіл блоків за вибраною технікою хешування.

$$\text{Computation}=\frac{\text{Keys}_{\text{Dataset}_1\text{HashedBucket}_n}\cdot\text{Keys}_{\text{Dataset}_2\text{HashedBucket}_n}}{\sum_{n=B_1}^{B_n}\left(\text{Keys}_{\text{Dataset}_1\text{HashedBucket}_n}\cdot\text{Keys}_{\text{Dataset}_2\text{HashedBucket}_n}\right)}. \quad (3.9)$$

Оцінка F1. Метрика, що використовується для вимірювання точності класифікації вибраної техніки хешування. Це середнє гармонійне значення точності (частка відповідних екземплярів серед вилучених екземплярів) і відкликання (частка відповідних екземплярів, отриманих серед загальної кількості відповідних екземплярів).

$$\text{F1Score}=2\cdot\left(\frac{\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}}\right). \quad (3.10)$$

Метрика подібності косинусів, яка використовується для обчислення подібності між хешованими векторами шляхом вимірювання косинуса кута між ними при проектуванні в багатовимірному просторі.

$$\text{F1Score}=2\cdot\left(\frac{\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}}\right). \quad (3.11)$$

4 АНАЛІЗ МЕТОДІВ ХЕШУВАННЯ MD5, SHA-1 ТА SHA-256

4.1 Показники оцінки виконання

Втрата працездатності. Цей показник використовується для визначення втрати продуктивності як часу виконання та відсотка ЦП. Втрата продуктивності, розрахована як різниця між максимальним і мінімальним значенням, потім розділена на максимальне значення. Ці показники обчислюються, щоб визначити, які параметри призвели до втрати продуктивності. Обчислення проводиться за формулою:

$$Lp=\frac{Min_{HashingMethod}-Max_{HashingMethod}}{Max_{HashingMethod}}\cdot 100\%, \quad (4.1)$$

де Lp – коефіцієнт втрати продуктивності;

$Min_{HashingMethod}$ – мінімальне значення методів хешування,

$Max_{HashingMethod}$ – максимальне значення методів хешування.

Щоб визначити найкращу продуктивність потрібно знайти найкращі алгоритми продуктивності (шифрування та хешування) за середнім часом і середнім процесором, розрахованим шляхом додавання параметрів, а потім розділу цих параметрів на число.

4.2 Результати експериментів

Було використано Visual Studio C#, де було реалізовано три хеш-функції (MD5, SHA-1 і SHA256) і два алгоритми шифрування (DES і 3DES).

Програма спочатку завантажує файл різного розміру, а потім передає його в MD5, бере значення ключа і передає його одному з алгоритмів шифрування. Цей процес повторювався для всіх шифрувань і хеш-функцій, а потім порівнювалися результати, отримані в кожному випадку.

Сценарій перший – шифрування хешуванням. Було отримано дайджест розміром – 32 МБ для MD5, 80 МБ для SHA-1 і 80 МБ для SHA-256. І в кожному

тесті обчислювався відсоток ЦП і час виконання до шифрування та після шифрування. Мета полягала в тому, щоб знайти, який із попередніх алгоритмів шифрування потребуватиме більше відсотків (використання ЦП) і часу виконання

Етапи виконання хешування та шифрування представлені в таблиці 4.1.

Таблиця 4.1 – Етапи виконання хешування та шифрування.

Номер фази	Метод хешування	Метод шифрування
Фаза 1	MD5	DES
		3DES
Фаза 2	SHA-1	DES
		3DES
Фаза 3	SHA-256	DES
		3DES

Порівнювалися результати на фазі 1, фазі 2 і фазі 3, щоб знайти найкращу продуктивність у середньому відсотку ЦП і середньому часу виконання. Результати показують, яка комбінація між хеш-функцією та алгоритмом шифрування є кращою з точки зору продуктивності підрахунку втрати продуктивності.

4.3 Метод хешування MD5

Цей алгоритм приймає різні розміри, але дає фіксований розмір. Було взято експеримент № 1 як приклад. Цей процес використовує MD5 з розміром файлу 10 Кбайт перед шифруванням. Час виконання до шифрування становив 49 мс. Відсоток ЦП до шифрування становив 1,190%. Результати навантаження процесору при роботі MD5 представлені в таблиці 4.2. Результати виконання MD5 перед алгоритмами хешування показано на рис.4.1.

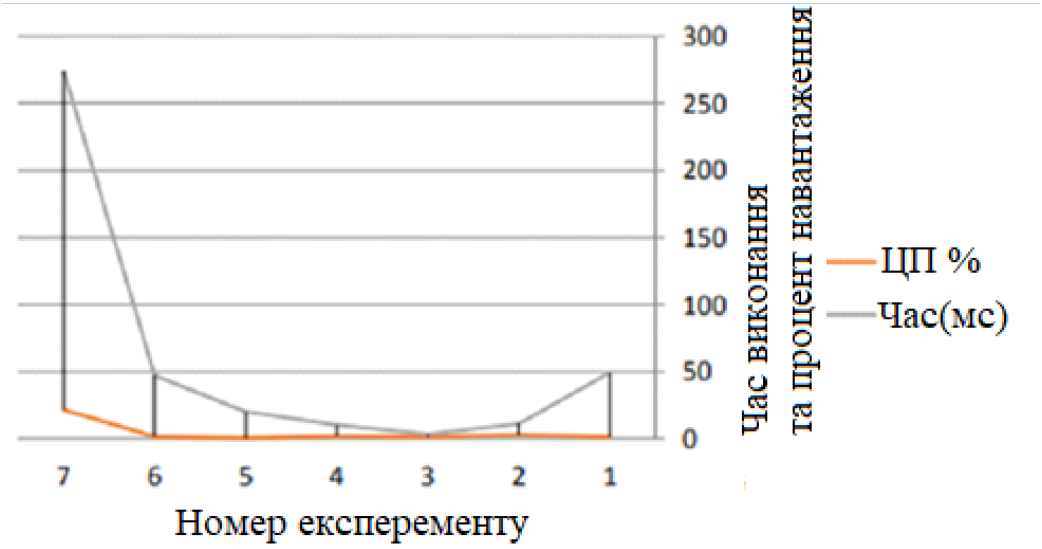

Рисунок 4.1 – Результати MD5 перед алгоритмами шифрування

Таблиця 4.2 – Результати MD5

Номер експерименту	Розмір файлу, Кб	Назва хеш-функції	Хеш-дайджест	CPU, %	Час (мс)
1	10	MD5	32	1,190	49
2	100	MD5	32	2,623	11
3	500	MD5	32	1,318	3
4	1500	MD5	32	1,727364	10
5	3	MD5	32	0,41	20
6	10	MD5	32	1,482	47
7	100	MD5	32	21,519	274
Середнє значення				42,324	59,143

З таблиці 4.2 і рис. 4.1 видно, що відсоток ЦП збільшувався з розміром файлу. І час виконання збільшувався з розміром файлу. Для їх результату понад 500 КБ.

Загальний час для семи експериментів становив 414 мс, а загальний процесор для семи експериментів 30,27%, тоді як середній час хешування всіх експериментів дорівнює 59,143 мс, а середній відсоток хешування ЦП дорівнює 42,3%.

Розмір ключа був одним із параметрів, які необхідно було дослідити, але було визначено, що деякі методи шифрування та хешування не дозволяють обрати розмір ключа. Для цього розмір ключа був дуже малим, і точність аналізу була

поганою з цих причин. Таблиця 4.3 показує відсоток навантаження процесору при шифруванні дайджесту 32 за допомогою DES і 3DES для кількох експериментів.

Для уточнення результатів у попередній таблиці на рис. 4.2 зображений відсоток навантаження процесору при використанні алгоритмів шифрування DES і 3DES з хешем MD5.

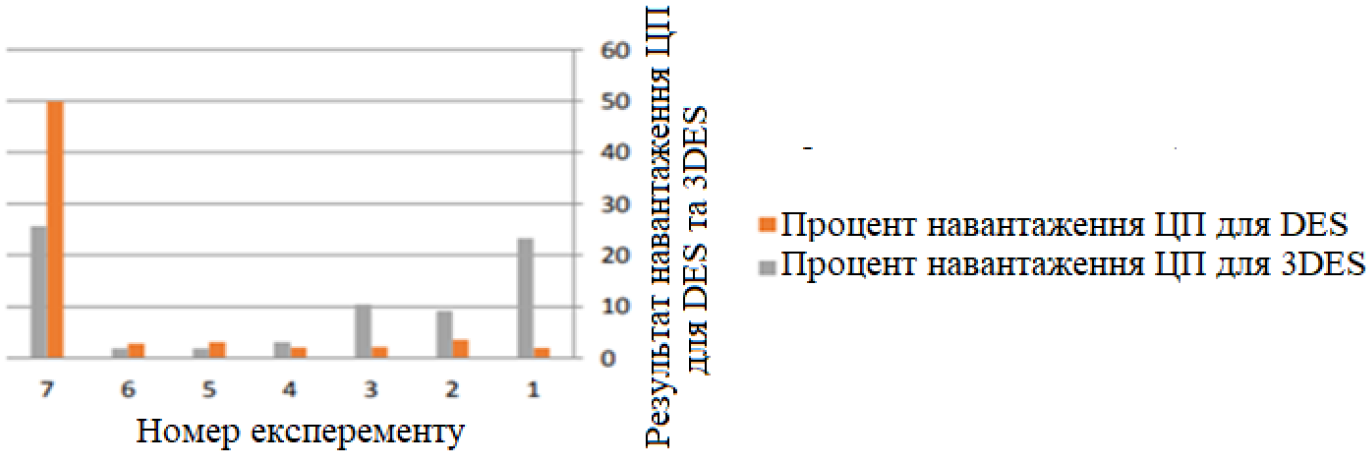

Рисунок 4.2 – Відсоток навантаження процесору

Таблиця 4.3 – Результати навантаження процесору при хешуванні методом MD5 та шифруванні за допомогою DES і 3DES

Номер експерименту	Відсоток CPU% DES	Відсоток CPU % 3DES
1	2,00	23,33
2	3,56	9,16
3	2,15	10,44
4	2,093	3,08
5	3,064	1,895
6	2,79	1,91
7	49,956	25,699
Середнє значення	9,37	10,79

На рис. 4.2 показано що для експериментів (1, 2, 3 і 4) відсоток ЦП при використанні DES менше, ніж при використанні 3DES. Для експериментів (5, 6 і 7) відсоток ЦП при використанні 3DES менше, ніж DES. Також на рис. показано, що продуктивність CPU для DES краща, ніж потрійний DES. Оскільки нижчий

відсоток ЦП є більшим, ніж вищий відсоток ЦП. На рис. 4.3 показано середній відсоток навантаження процесору для шифрування з хешуванням функцією MD5.

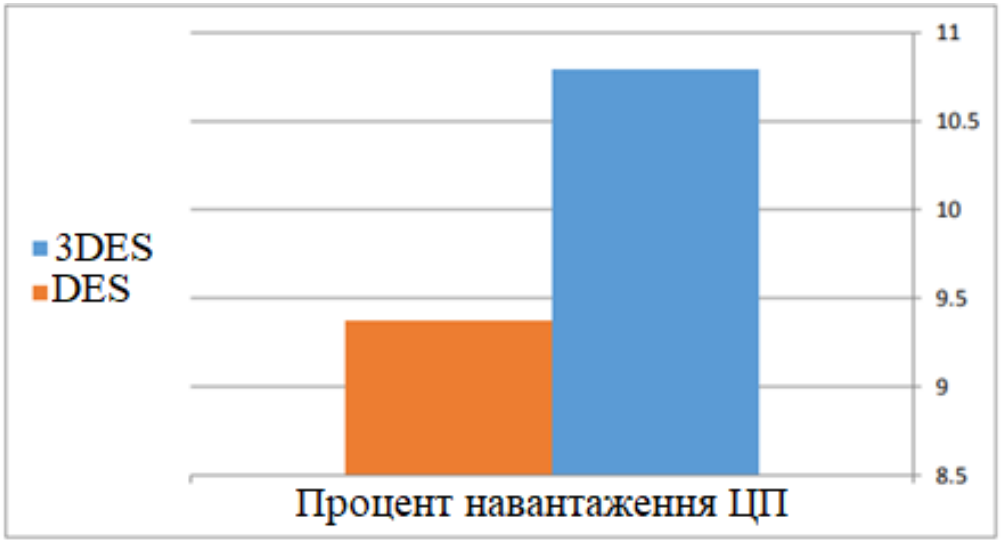

Рисунок 4.3 – Середній відсоток навантаження процесору при хешуванні функцією MD5

На рис. 4.3 зображений середній відсоток навантаження процесору при шифруванні DES і 3DES. Рис. 4.3 показує, що середній відсоток продуктивності CPU для DES кращий, ніж 3DES. У таблиці 4.4 показано час шифрування для змінного розміру файлу та використання MD5.

Таблиця 4.4 – Результати часу виконання алгоритму MD5

Номер експерименту	Час виконання (мс) DES	Час виконання (мс) 3DES
1	96	54
2	13	13
3	5	4
4	12	12
5	24	22
6	49	49
7	274	276
Середнє значення	67,57	61,43

Для уточнення результати в попередній таблиці наведено на рис. 4.4 представлений час виконання шифрування для алгоритмів шифрування DES і 3DES з хешем MD5.

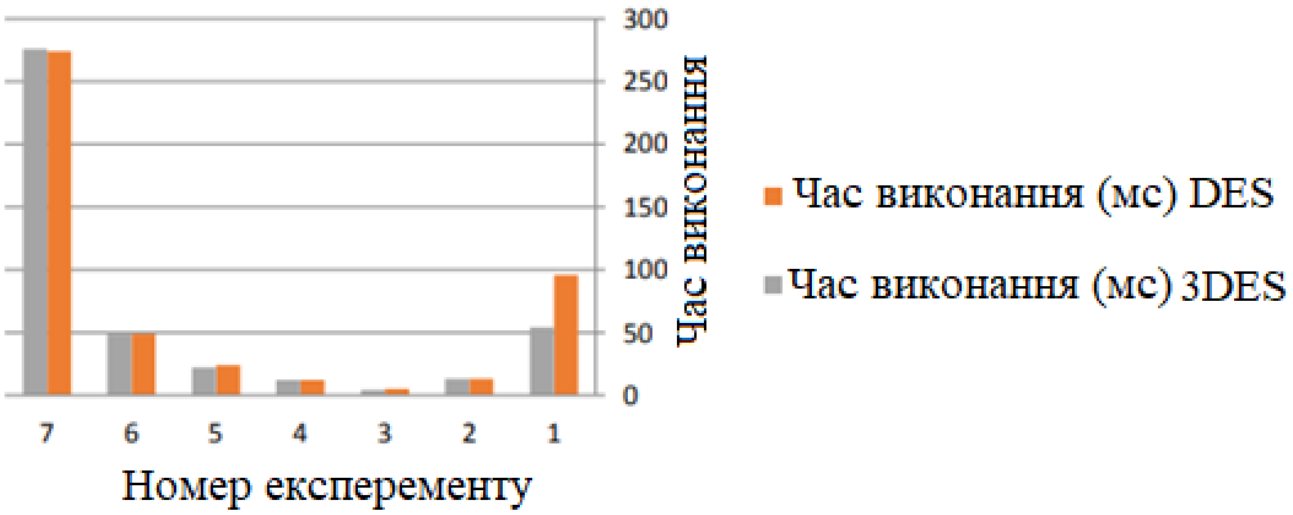

Рисунок 4.4 – Час виконання шифрування

На рис. 4.4 показано, що для експерименту (1) час виконання для 3DES менше ніж для DES. Для експерименту (3) час виконання для 3DES менше, ніж для DES. Час виконання експерименту (5) для 3DES менше, ніж для DES. Час виконання експерименту (7) для DES менше, ніж для 3DES. На рис. 4.5 представлений середній час виконання шифрування для алгоритмів шифрування DES і 3DES з хешем MD5.

Рисунок 4.5 – Середній час виконання шифрування

На рис. 4.5 видно середній час виконання шифрування для DES і 3DES. Також на рис. 4.5 показано, що продуктивність середнього часу виконання для 3DES краща, ніж DES. Оскільки чим менший час виконання, тим краще, тим вище відсоток навантаження процесору.

В таблиці 4.5 представлений відсоток навантаження процесору після шифрування хеш-функцією MD5, у якої параметром є розмір ключа – який і треба було дослідити.

Таблиця 4.5 – Підсумований результат MD5 для відсотка навантаження процесору після шифрування різними ключами.

Підсумований відсоток ЦП	DES	3DES
Після шифрування ключем	9,37	10,79

Було відзначено, що багато методів хешування та шифрування дозволяють обирати розмір ключа. Тому вплив розміру ключа був дуже малим, і точність догляду була невеликою. В результаті було виявлено що DES краще, ніж 3DES.

Сценарій 2 (хешування над зашифрованими даними). Програма об’єднує текст різного розміру з хешем результату MD5, бере значення ключа, потім передає його одному з алгоритмів шифрування. Цей процес повторюється для всіх шифрувань і хеш-функцій, а потім порівнюються результати, отримані в кожному випадку.

І в кожному тесті обчислюється відсоток навантаження процесору і час виконання до шифрування і після шифрування алгоритмам буде потрібно більше відсотків (використовувати ЦП) і часу виконання. Більше часу потрібно для шифрування.

В таблиці 4.6 представлено етапи виконання алгоритму між хешуванням злиття з відкритим текстом і шифруванням. Було порівняно отримані результати на фазі 1, фазі 2 і фазі 3, щоб знайти найкращу продуктивність у середньому відсотку ЦП і середньому часу виконання. Отримані результати покажуть, яка комбінація між хеш-функцією та алгоритмом шифрування є кращою з точки зору безпеки під час обчислення втрати продуктивності.

Таблиця 4.6 Етап виконання між хешуванням злиття з відкритим текстом і шифруванням.

Фаза 1	Результат злиття MD5 із простим текстом	DES
		3DES
Фаза 2	Результат злиття SHA-1 із простим текстом	DES
		3DES
Фаза 3	Результат злиття SHA-256 із простим текстом	DES
		3DES

В таблиці 4.7 показано відсоток ЦП і час хешування для змінного розміру файлу за допомогою результату злиття MD5 з іншим розміром відкритого тексту перед шифруванням, таким як DES. В таблиці 4.7 представлений результат хеш функції з різним розміром даних перед шифруванням.

Таблиця 4.7 – Зразок злиття результату хешу з даними різного розміру

Номер експерименту	Ім'я хеша	Розмір результату злиття	CPU %	Час(мс)
1	MD5	10,476 Кб	30,31	3053
2	MD5	100,868 Кб	0,87	3021
3	MD5	500,298 Кб	56,42	3035
4	MD5	1500,016 Кб	0,75	3070
5	MD5	3,072 Мб	53,08	3135
6	MD5	10,239154 Мб	22,34	3371
7	MD5	100,399904 Мб	27,28	6352
Середнє значення			27,29286	3576,71

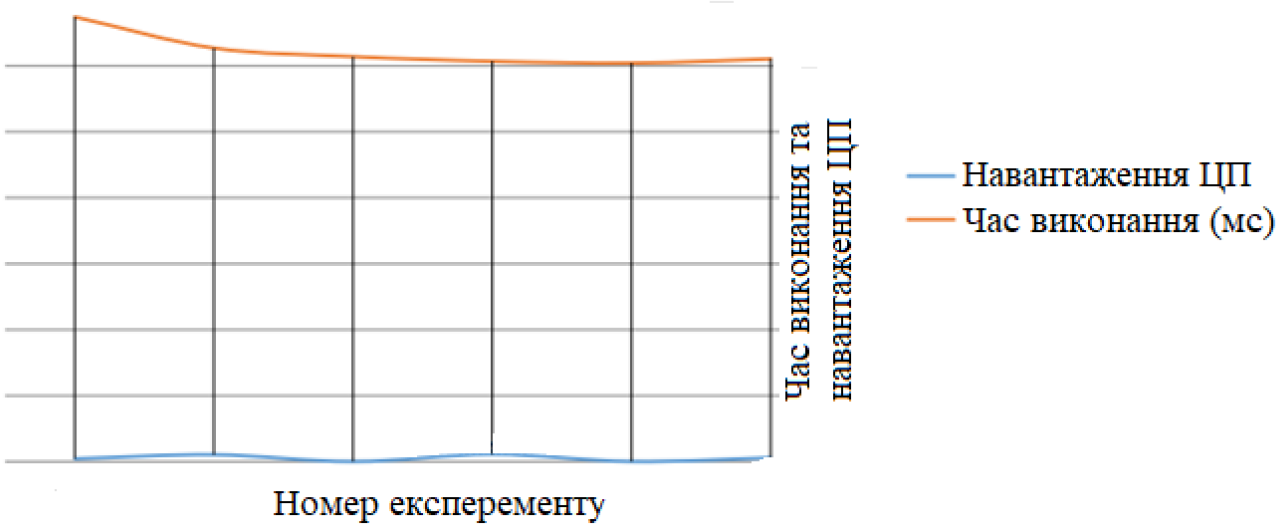

Рисунок 4.6 – Результат (злиття) для часувиконання та процент навантаження процесору перед шифруванням

З таблиці 4.7 та рис 4.6 видно, що час виконання збільшувався з розміром файлу, а навантаження процесора збільшилося на значення вище 56.

Було обрано розмір ключа, який був одним із параметрів, які досліджувались, та було визначено, що деякі методи шифрування та хешування не дозволяють нам вибрати розмір ключа. Для цього дослідження розмір ключа був дуже малим, і точність дослідження була поганою через це. Таблиця 4.8, показує відсоток навантаження процесору при шифруванні із з'єднанням з різними розмірами файлу хеш функцією MD5.

Таблиця 4.8 – Процент навантаження процесору при шифруванні із з'єднанням з різними розмірами файлу хеш функцією MD5

Номер експерименту	CPU% DES	CPU % 3DES
1	3,8,9	22,963
2	4,121	8,021
3	7,948	28,55
4	23,899	3,207
5	27,271	4,097
6	49,941	0,568
7	19,49817	18,924
Середнє значення	19,49817	12,33286

На рис. 4.7 показано відсоток навантаження процесору для алгоритмів шифрування DES і 3DES результату хешування об’єднаного з відкритим текстом.

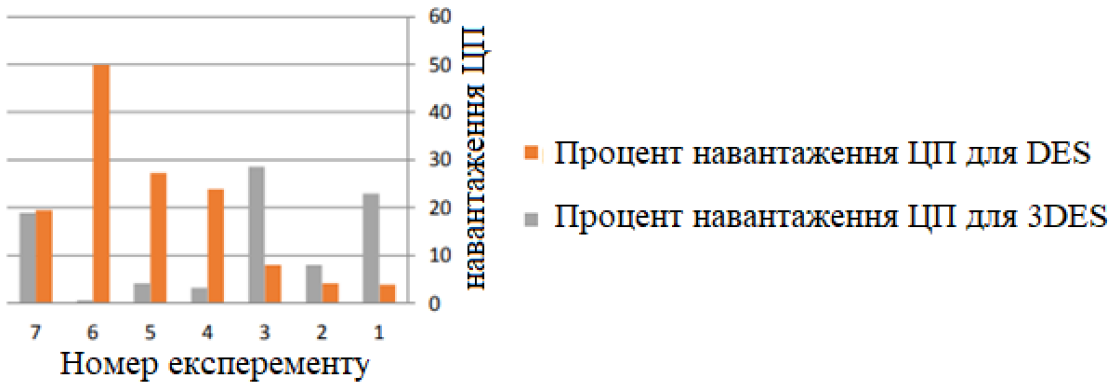

Рисунок 4.7 – Відсоток навантаження процесору при шифруванні хеш-суми об’єднаної з відкритим текстом

На рис. 4.7 показано, що в експериментах (1, 2 і 3) відсоток навантаження процесору для DES менше, ніж для 3DES. Для експериментів (4, 5, 6 і 7) відсоток навантаження процесору для 3DES менше, ніж DES. На рис. 4.7 показано, що продуктивність ЦП для 3DES краща за DES.

На рис. 4.8 показано середнє значення відсотку навантаження процесору.

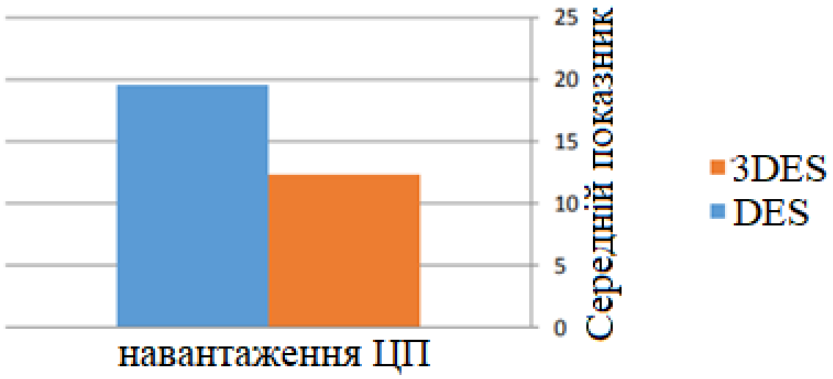

Рисунок 4.8 – Середнє значення відсотку навантаження процесору

На рис. 4.8 відображене середнє значення часу виконання шифрування для DES і 3DES. На рис. 4.8 показано, що середнє значення продуктивності у відсотках CPU для 3DES краща, ніж DES. Оскільки нижчий відсоток ЦП краще, ніж вищий відсоток ЦП. В таблиці 4.9 представлений час виконання шифрування.

Таблиця 4.9 – Час виконання шифрування

Номер експерименту	Час виконання (м,с) з DES	Час виконання (м,с) з DES
1	31	59
2	30	15
3	30	5
4	31	14
5	32	24
6	33	51
7	69	278
Середнє значення	36,6	63,71

Для уточнення результатів попередньої таблиці, на рис. 4.9 відображене порівняння часу виконання шифрування

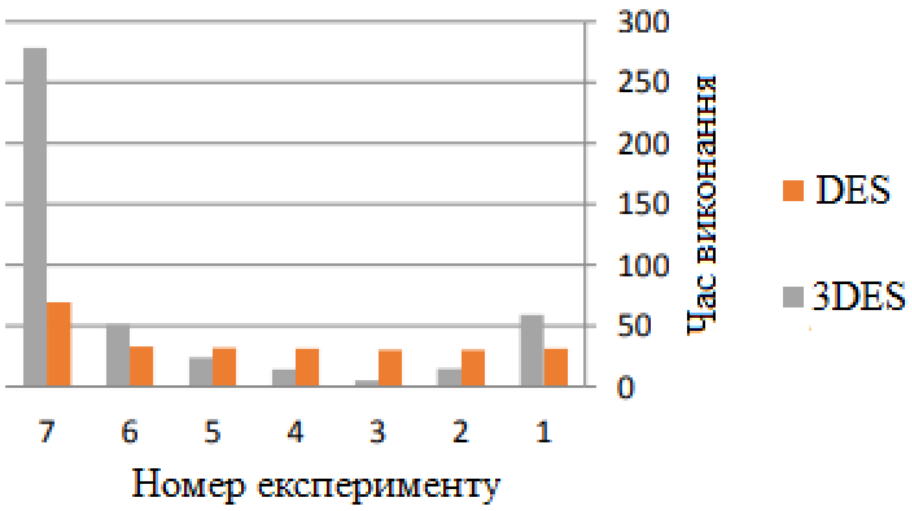

Рисунок 4.9 – Час виконання злиття MD5

На рис. 4.9 показано час виконання шифрування для алгоритмів шифрування DES і 3DES результату хеш-функції та відкритим текстом. На рис. 4.9 показано, що для експериментів (1, 6 і 7) час виконання для DES менше, ніж 3DES.

На рис. 4.10 показаний середній час виконання шифрування для алгоритмів шифрування DES і 3DES з хешем MD5.

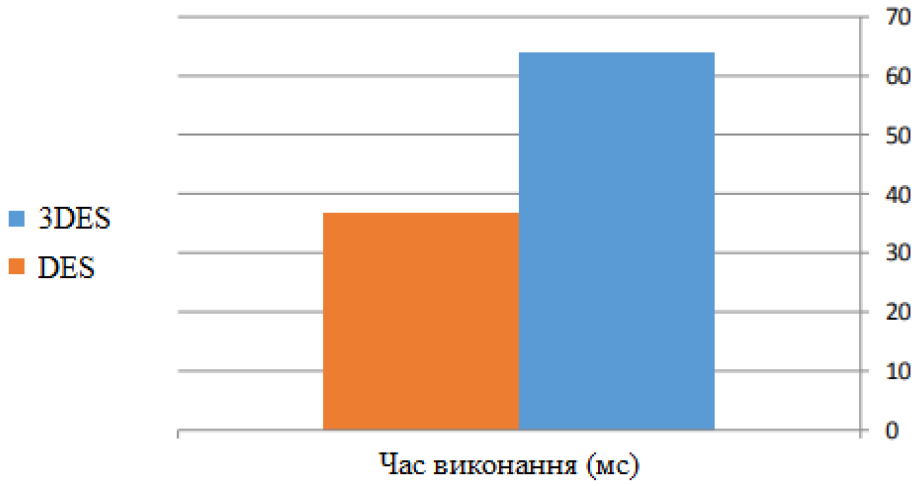

Рисунок 4.10 – Середній час виконання злиття MD5.

4.4 Метод хешування SHA-1

Таблиця 4.10 показує підсумований середній час виконання результату та відсоток навантаження процесору перед шифруванням при хешуванні шифрування сценарію та хешуванні сценарію над зашифрованими даними.

Таблиця 4.10 – підсумований результат перед шифруванням

Час виконання до шифрування (сценарій 1)	Відсоток ЦП до шифрування (сценарій 1)
456,6	31,72

Таблиця 4.11 показує відсоток навантаження процесору при хешуванні алгоритмом SHA-1.

Таблиця 4.11 – Підсумований відсоток навантаження процесору під час шифрування.

Підсумуйте відсоток ЦП (після шифрування хешуванням)	DES	3DES
Після шифрування	72,72	72,72

Таблиця 4.12 показує хеш результату SHA-1 після шифрування з різними ключами. Результатом було що DES та 3DES мали однаковий час виконання.

Таблиця 4.12 – Підсумований час виконання шифрування з різними ключами.

Підсумований час виконання (після шифрування хешуванням)	DES	3DES
Після шифрування	90,73	90,73

Таблиця 4.13 показує процент навантаження процесору під час з’єднання результату хеш-функції SHA-1 з простим текстом. Як можна побачити на таблиці, DES має трішки кращій показник, ніж 3DES.

Таблиця 4.13 – Підсумований результат навантаження процесору під час злиття результату хеш функції SHA-1 з простим текстом.

Підсумований відсоток ЦП (злиття)	DES	3DES
Після шифрування	82,26	82,39

В таблиці 4.14 можна побачити час виконання з’єднання результату хеш-функції з простим текстом. Як можна побачити, DES має кращий показник у часі, ніж 3DES.

Таблиця 4.14 – Підсумований результат часу виконання з’єднання після шифрування різними ключами.

Підсумований час виконання (м,с) (Об’єднання)	DES	3DES
Після шифрування	72,72	1284,47

4.5 Метод хешування SHA-256

Таблиця 4.15 показує середній час виконання хешування та відсоток навантаження процесору перед шифруванням та хешуванні зашифрованих даних.

Таблиця 4.15 – Підсумований результат перед шифруванням

Час хешування до шифрування	Відсоток навантаження процесору до шифрування
169,94	42,86

Таблиця 4.16 показує результат навантаження процесору при хешуванні функцією SHA-256 після шифрування. Можна побачити що 3DES навантажує ЦП трішки менше, ніж DES.

Таблиця 4.16 – Підсумований відсоток навантаження процесору після шифрування.

Підсумований відсоток ЦП	DES	3DES
Після шифрування	93,66	85,93

Таблиця 4.17 показує підсумковий час виконання хешування функцією SHA-256 після шифрування за допомогою DES та 3DES. Як можна побачити у таблиці, алгоритм шифрування 3DES має набагато менший час виконання, ніж алгоритм шифрування DES

Таблиця 4.17 – Підсумований час виконання хешування після шифрування.

Підсумований час виконання	DES	3DES
Після шифрування	3669,20	246,24

Таблиця 4.18 показує підсумковий відсоток навантаження процесору під час хешування функцією SHA-256 після шифрування з додаванням простого тексту за допомогою DES та 3DES. Як можна побачити у таблиці, алгоритм шифрування 3DES навантажує ЦП трішки сильніше, ніж алгоритм шифрування DES.

Таблиця 4.18 – Підсумований результат відсотка навантаження процесору під час хешування після шифрування з додаванням простого тексту.

Підсумований відсоток ЦП	DES	3DES
Після шифрування	87,50	89,01

Таблиця 4.19 показує підсумковий час виконання хешування функцією SHA-256 після шифрування з додаванням простого тексту за допомогою DES та 3DES. Як можна побачити у таблиці, алгоритм шифрування 3DES має час виконання більший, ніж алгоритм шифрування DES.

Таблиця 4.19 – Підсумований час виконання результату після шифрування з додаванням простого тексту.

Підсумуйте час виконання	DES	3DES
Після шифрування	1482,42	3874,07

Було порівняно результати, отримані в алгоритмі шифрування MD5, з результатами, отриманими в SHA-1 і з SHA-256 у відсотках навантаження процесору і часу виконання.

Результат в таблиці 4.20 відображає процент навантаження процесору відносно методів хешування та алгоритмів шифрування.

Таблиця 4.20 – Порівняння проценту навантаження процесору між методами хешування після шифрування

Методи хешування	DES	3DES
MD5	9,37	10,79
SHA-1	72,72	72,72
SHA-256	93,66	85,93

Згідно з результатами, що представлені в таблиці, процент навантаження процесору, при використанні методу хешування SHA-256 та шифруванні за допомогою DES, помітно більший, в порівняні з використанням методів хешування

MD5 та SHA-1. За формулою (4.1) знаходимо коефіцієнт втрати продуктивності Lp = (93,66-9,37)/ 93,66= 89,995%.

При шифруванні за допомогою 3DES складається таж сама картина – при використанні методу хешування SHA-256, процент навантаження процесору помітно більший, ніж при використанні методів MD5 та SHA-1. Коефіцієнт втрати продуктивності складає Lp = (85,93-10,79)/ 85,93= 87,44%.

Дані таблиці 4.21 відображають відсотка навантаження процесору відносно методів хешування та алгоритмів шифрування з додаванням простого тексту.

Таблиця 4.21 – Порівняння відсотка навантаження процесору між методами хешування після шифрування з додаванням простого тексту.

Методи хешування	DES	3DES
MD5	19,498	12,333
SHA-1	82,26	82,39
SHA-256	87,5	89,01

Згідно з результатами, що представлені в таблиці, процент навантаження процесору, при використанні методу хешування SHA-256 та шифруванні за допомогою DES, помітно більший, в порівняні з використанням методів хешування MD5 та SHA-1. За формулою (4.1) знаходимо коефіцієнт втрати продуктивності Lp = (87,50-19,498)/87,50= 77,72%.

При шифруванні за допомогою 3DES складається таж сама картина – при використанні методу хешування SHA-256, процент навантаження процесору помітно більший, ніж при використанні методів MD5 та SHA-1. Коефіцієнт втрати продуктивності складає Lp = (89,01-12.333)/89,01= 86,14%.

Результат в таблиці 4.22 відображає час виконання відносно методів хешування та алгоритмів шифрування.

Таблиця 4.22 – Порівняння часу виконання між методами хешування після шифрування

Методи хешування	DES	3DES
MD5	67,57	61,43
SHA-1	90,73	90,73
SHA-256	3669,20	246,24

Згідно з результатами, що представлені в таблиці, час виконання, при використанні методу хешування SHA-256 та шифруванні за допомогою DES, помітно більший, в порівняні з використанням методів хешування MD5 та SHA-1. За формулою (4.1) знаходимо коефіцієнт втрати продуктивності Lp = (3669,20-67,57)/ 3669,20= 98,19%.

При шифруванні за допомогою 3DES складається таж сама картина – при використанні методу хешування SHA-256, час виконання помітно більший, ніж при використанні методів MD5 та SHA-1. Коефіцієнт втрати продуктивності складає Lp = (246,24-61,43)/246,24= 75,05%.

Результат в таблиці 4.23 відображає час виконання відносно методів хешування та алгоритмів шифрування з додаванням простого тексту.

Таблиця 4.23 – Порівняння часу виконання між методами хешування після шифрування з додаванням простого тексту

Методи хешування	DES	3DES
MD5	1170,11	63,71
SHA-1	72,72	1284,47
SHA-256	1482,42	3874,07

Згідно з результатами, що представлені в таблиці, час виконання, при використанні методу хешування SHA-256 та шифруванні за допомогою DES, помітно більший, в порівняні з використанням методів хешування MD5 та SHA-1. За формулою (4.1) знаходимо коефіцієнт втрати продуктивності Lp = (1482,42-72,72)/ 1482,42= 95,09%.

При шифруванні за допомогою 3DES складається таж сама картина – при використанні методу хешування SHA-256, час виконання помітно більший, ніж при використанні методів MD5 та SHA-1. Коефіцієнт втрати продуктивності складає Lp = (3874.07-63.71)/3874.07= 98.36%.

ВИСНОВКИ

В роботі був проведений аналіз продуктивності методів хешування для двох сценаріїв – до шифрування та після шифрування. Для оцінювання було обрано два показники – час виконання хешування та процент навантаження процесору під час шифрування.

В обох сценаріях, до шифрування та після шифрування, алгоритм хешування MD5 показав кращі результати.

Для сценарію, в якому хешування відбувалося до шифрування, процент навантаження та час виконання – однакові, оскільки всі методи хешування використовували ключі однакого розміру.

Для сценарію, в якому хешування відбувалося після шифрування, кращий час виконання був після шифрування алгоритмом 3DES. Середній відсоток навантаження процесору був кращим після шифрування алгоритмом DES.

Також існує ще декілька рекомендацій для майбутніх робот на цю тему. По-перше, звісно можна обрати інші методи хешування з ціллю отримання більш об’єктивної оцінки та знаходження більш оптимального методу хешування з точки зору навантаження процесору та часу виконання. Також можна обрати методи, які використовують більші розміри ключів. Та можна розглянути додатковий параметр продуктивності додатково до параметрів які були розглянуті в цій дипломній роботі. Наприклад, за параметр можна обрати ОЗП.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Сахаров М. Г. Ефективні методи та засоби захисту фішингових атак / М. Г Сахаров, А. О. Яіцький – Варшава: Problems of science and practice, tasks and ways to solve them. – 2022. – С. 337-339

2. Сахаров М. Г. Застосування хмарних технологій в космосі / М. Г. Сахаров, А. О. Яіцький – Київ: Priority directions of science and technology development. – 2021. – С. 417-419

3. Ghemawat S. The google file system / S. Ghemawat, H. Gobioff, S.-T. Leung – 2003.

4. Borthakur D. The hadoop distributed file system: Architecture and design / D. Borthakur // Hadoop Project Website – 2007 – С. 21.

5. F. Chang F. Bigtable: A distributed storage system for structured data / F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber // ACM Transactions on Computer Systems (TOCS) – 2008 – С. 4.

6. Begoli E. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources / E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, D. Lemire // International Conference on Management of Data – 2018 – С. 221–230.

7. Armbrust M. Spark sql: Relational data processing in spark / M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi // ACM SIGMOD international conference on management of data – 2015. – С. 1383–1394.

8. Salloum S. Big data analytics on apache spark / S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang // International Journal of Data Science and Analytics – 2016 – С. 145–164.

9. Zaharia M. Discretized streams: Fault-tolerant streaming computation at scale. / M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica // ACM symposium on operating systems principles – 2013 – С. 423–438.

10. Chilimbi T. M. Project adam: Building an efficient and scalable deep learning training system / T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman // OSDI – 2014 – С. 571–582.

11. Rodrigues M. Describing and comparing big data querying tools / M. Rodrigues, M. Y. Santos, J. Bernardino // World Conference on Information Systems and Technologies – 2017 – С. 115–124.

12. Bu Y. Hadoop: efficient iterative data processing on large clusters / Y. Bu, B. Howe, M. Balazinska, M. D. Ernst // VLDB Endowment – 2010 – С. 285–296.

13. Li. W. -J. Feature learning based deep supervised hashing with pairwise labels / W.-J. Li, S. Wang, W.-C. Kang // Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16) – 2015 – С. 1711–1717.

14. Wang X. Deep supervised hashing with triplet labels / X. Wang, Y. Shi, K. M. Kitani // Asian conference on computer vision – 2016 – С. 70–84.

15. Zhang P. Supervised hashing with latent factor models / P. Zhang, W. Zhang, W.-J. Li, M. Guo // 37th international ACM SIGIR conference on Research & development in information retrieval – 2014 – С. 173–182

16. Zhu H. Deep hashing network for efficient similarity retrieval / H. Zhu, M. Long, J. Wang, Y. Cao // Thirtieth AAAI Conference on Artificial Intelligence – 2016 – С. 2415–2421.

17. Li. Q. Deep supervised discrete hashing / Q. Li, Z. Sun, R. He, T. Tan // Advances in neural information processing systems – 2017 – С. 2482–2491

18. Braams B. Predicate Pushdown in Parquet and Apache / B. Braams // PhD thesis, Universiteit van Amsterdam – 2018.

19. Yao X. Big spatial vector data management: a review / X. Yao, G. Li // Big Earth Data – 2018 – С. 108–129.

20. Svensson P. Emerging database systems in support of scientific data / P. Svensson, P. Boncz, M. Ivanova, M. Kersten, N. Nes, D. Rotem // Scientific Data Management: Challenges Technology and Deployment – 2010 – С. 235–277.

21. Wang J. Learning to hash for indexing big data – a survey / J. Wang, W. Liu, S. Kumar, S.-F. Chang // IEEE – 2015 – С. 34–57.

22. Doersch C. What makes Paris look like Paris? / C. Doersch, S. Singh, A. Gupta, J. Sivic, A. A. Efros // Communications of the ACM – 2015 – С. 103–110.

23. Yoon C. E. Earthquake detection through computationally ecient similarity search / C. E. Yoon, O. O'Reilly, K. J. Bergen, G. C. Beroza // Science Advances – 2015.