ДОДАТОК А

Приклади кодів програм

А.1 Код Геш-таблиці з методом відкритої адресації

```
public class RedisLikeHashMap<K, V> implements HashTable <K, V> {
    private static final int INITIAL_CAPACITY = 16;
    private static final float LOAD_FACTOR = 0.75f;
    private Entry<K, V>[] table;
    private int size;

    public RedisLikeHashMap() {
        table = new Entry[INITIAL_CAPACITY];
    }


    private static class Entry<K, V> {
        final K key;
        V value;

        Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }
    }

    private int hash(Object key) {
        return Math.abs(key.hashCode() % table.length);
    }

    public void put(K key, V value) {
        if (size >= LOAD_FACTOR * table.length) {
            resize();
        }

        int index = hash(key);
        while (table[index] != null && !table[index].key.equals(key))
{
            index = (index + 1) % table.length;
        }

        if (table[index] == null) {
            size++;
        }
        table[index] = new Entry<>(key, value);
    }

    public V get(K key) {
        int index = hash(key);
        while (table[index] != null) {
            if (table[index].key.equals(key)) {
                return table[index].value;
            }
            index = (index + 1) % table.length;
        }
```

```
            return null;
        }

        public V remove(K key) {
            int index = hash(key);
            while (table[index] != null) {
                if (table[index].key.equals(key)) {
                    V value = table[index].value;
                    table[index] = null;
                    size--;
                    rehash(index);
                    return value;
                }
                index = (index + 1) % table.length;
            }
            return null;
        }



    }
```

А.2 Код Геш-таблиці з методом бакетів

```
    import java.util.TreeMap;

    public class RedisLikeHashMapWithTree<K, V> implements HashTable <K,
V> {
        private static final int INITIAL_CAPACITY = 16;
        private static final float LOAD_FACTOR = 0.75f;
        private TreeMap<K, V>[] table;
        private int size;

        public RedisLikeHashMapWithTree() {
            table = new TreeMap[INITIAL_CAPACITY];
            for (int i = 0; i < INITIAL_CAPACITY; i++) {
                table[i] = new TreeMap<>();
            }
        }

        private int hash(Object key) {
            return Math.abs(key.hashCode() % table.length);
        }

        public void put(K key, V value) {
            if (size >= LOAD_FACTOR * table.length) {
                resize();
            }

            int index = hash(key);
            if (table[index].put(key, value) == null) {
                size++;
            }
        }

        public V get(K key) {
            int index = hash(key);
```

```
        return table[index].get(key);
    }

    public V remove(K key) {
        int index = hash(key);
        V value = table[index].remove(key);
        if (value != null) {
            size--;
        }
        return value;
    }

    private void resize() {
        TreeMap<K, V>[] oldTable = table;
        table = new TreeMap[oldTable.length * 2];
        for (int i = 0; i < table.length; i++) {
            table[i] = new TreeMap<>();
        }
        size = 0;

        for (TreeMap<K, V> bucket : oldTable) {
            for (K key : bucket.keySet()) {
                put(key, bucket.get(key));
            }
        }
    }
    public int size() {
        return size;
    }
}
```

A.3 Код Б-дерево

```
public class BTree<T extends Comparable<T>> {

    private int minKeySize = 1
    private int minChildrenSize = minKeySize + 1; /
    private int maxKeySize = 2 * minKeySize
    private int maxChildrenSize = maxKeySize + 1;

    private Node<T> root = null;
    private int size = 0;

    public BTree() {
    }

    public BTree(int order) {
        this.minKeySize = order;
        this.minChildrenSize = minKeySize + 1;
        this.maxKeySize = 2 * minKeySize;
        this.maxChildrenSize = maxKeySize + 1;
    }

    public boolean add(T value) {
        if (root == null) {
            root = new Node<T>(null, maxKeySize, maxChildrenSize);
            root.addKey(value);
```

```
            } else {
                Node<T> currentNode = root;
                while (currentNode != null) {
                    if (currentNode.numberOfChildren() == 0) {
                        currentNode.addKey(value);
                        if (currentNode.numberOfKeys() <= maxKeySize) {
                            break;
                        }
                        split(currentNode);
                        break;
                    }
                    T lesserKey = currentNode.getKey(0);
                    if (value.compareTo(lesserKey) <= 0) {
                        currentNode = currentNode.getChild(0);
                        continue;
                    }

                    int numberOfKeys = currentNode.numberOfKeys();
                    int lastKeyIndex = numberOfKeys - 1;
                    T greaterKey = currentNode.getKey(lastKeyIndex);
                    if (value.compareTo(greaterKey) > 0) {
                        currentNode = currentNode.getChild(numberOfKeys);
                        continue;
                    }

                    for (int i = 1; i < currentNode.numberOfKeys(); i++)
{
                        T previousKey = currentNode.getKey(i - 1);
                        T nextKey = currentNode.getKey(i);
                        if (value.compareTo(previousKey) > 0 &&
value.compareTo(nextKey) <= 0) {
                            currentNode = currentNode.getChild(i);
                            break;
                        }
                    }
                }
            }

            size++;
            return true;
        }

        private T remove(T value, Node<T> node) {
            if (node == null) return null;

            T removedValue = null;
            int index = node.indexOf(value);
            removedValue = node.removeKey(value);
            if (node.numberOfChildren() == 0) {
                if (node.parent != null && node.numberOfKeys() <
minKeySize) {
                    this.combine(node);
                } else if (node.parent == null && node.numberOfKeys() ==
0) {
                    root = null;
                }
            } else {
                Node<T> lesserNode = node.getChild(index);
```

```
                Node<T> greatestNode = this.getGreatestNode(lesserNode);
                T replacementValue =
this.removeGreatestValue(greatestNode);
                node.addKey(replacementValue);
                if (greatestNode.parent != null &&
greatestNode.numberOfKeys() < minKeySize) {
                    this.combine(greatestNode);
                }
                if (greatestNode.numberOfChildren() > maxChildrenSize) {
                    this.split(greatestNode);
                }
            }

            size--;
            return removedValue;
        }

    }
```

А.4 Код Скіп-листа

```
    public class SkipList<K extends Comparable<K>, V> {
        private static final int MAX_LEVEL = 16;
        private  final  Node<K,  V>  head  =  new  Node<>(null,  null,
MAX_LEVEL);
        private final Random random = new Random();

        static class Node<K, V> {
            final K key;
            V value;
            final Node<K, V>[] forward;

            Node(K key, V value, int level) {
                this.key = key;
                this.value = value;
                this.forward = new Node[level + 1];
            }
        }

        public void put(K key, V value) {
            Node<K, V>[] update = new Node[MAX_LEVEL + 1];
            Node<K, V> x = head;

            for (int i = MAX_LEVEL; i >= 0; i--) {
                while       (x.forward[i]       !=       null       &&
x.forward[i].key.compareTo(key) < 0) {
                    x = x.forward[i];
                }
                update[i] = x;
            }

            x = x.forward[0];

            if (x != null && x.key.compareTo(key) == 0) {
                x.value = value;
            } else {
                int level = randomLevel();
```

```
                x = new Node<>(key, value, level);
                for (int i = 0; i <= level; i++) {
                    x.forward[i] = update[i].forward[i];
                    update[i].forward[i] = x;
                }
            }
        }

        public V get(K key) {
            Node<K, V> x = head;

            for (int i = MAX_LEVEL; i >= 0; i--) {
                while (x.forward[i] != null &&
x.forward[i].key.compareTo(key) < 0) {
                    x = x.forward[i];
                }
            }

            x = x.forward[0];

            if (x != null && x.key.compareTo(key) == 0) {
                return x.value;
            }

            return null;
        }

        public void remove(K key) {
            Node<K, V>[] update = new Node[MAX_LEVEL + 1];
            Node<K, V> x = head;

            for (int i = MAX_LEVEL; i >= 0; i--) {
                while (x.forward[i] != null &&
x.forward[i].key.compareTo(key) < 0) {
                    x = x.forward[i];
                }
                update[i] = x;
            }

            x = x.forward[0];

            if (x != null && x.key.compareTo(key) == 0) {
                for (int i = 0; i <= MAX_LEVEL; i++) {
                    if (update[i].forward[i] != x) break;
                    update[i].forward[i] = x.forward[i];
                }
            }
        }

        private int randomLevel() {
            int level = 0;
            while (random.nextInt(2) == 0 && level < MAX_LEVEL) {
                level++;
            }
            return level;
        }
```

А.5 Код графового сховища

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class GraphStore {
class Node {
    private final String id;
    private final Map<String, String> properties;
    private final List<Relationship> relationships;

    public Node(String id) {
        this.id = id;
        this.properties = new HashMap<>();
        this.relationships = new ArrayList<>();
    }
}

class Relationship {
    private final String type;
    private final Node startNode;
    private final Node endNode;
    private final Map<String, String> properties;

    public Relationship(String type, Node startNode, Node endNode) {
        this.type = type;
        this.startNode = startNode;
        this.endNode = endNode;
        this.properties = new HashMap<>();
    }
}

    private final Map<String, Node> nodes;
    private final Map<String, List<Relationship>> relationships;

    public GraphDatabase() {
        nodes = new HashMap<>();
        relationships = new HashMap<>();
    }

    public Node createNode(String id) {
        Node node = new Node(id);
        nodes.put(id, node);
        return node;
    }

    public Relationship createRelationship(String type, Node
startNode, Node endNode) {
        Relationship relationship = new Relationship(type, startNode,
endNode);
        startNode.addRelationship(relationship);
        endNode.addRelationship(relationship);
        relationships.computeIfAbsent(type, k -> new
ArrayList<>()).add(relationship);
        return relationship;
    }

    public Node getNode(String id) {
```

```
            return nodes.get(id);
        }

        public List<Relationship> getRelationships(String type) {
            return relationships.get(type);
        }

        public List<Node> findNodesByProperty(String key, String value) {
            List<Node> result = new ArrayList<>();
            for (Node node : nodes.values()) {
                if (value.equals(node.getProperty(key))) {
                    result.add(node);
                }
            }
            return result;
        }

        public List<Relationship> findRelationshipsByProperty(String key,
String value) {
            List<Relationship> result = new ArrayList<>();
            for (List<Relationship> relList : relationships.values()) {
                for (Relationship rel : relList) {
                    if (value.equals(rel.getProperty(key))) {
                        result.add(rel);
                    }
                }
            }
            return result;
        }

        public void deleteNode(String id) {
            Node node = nodes.remove(id);
            if (node != null) {
                for         (Relationship         rel         :         new
ArrayList<>(node.getRelationships())) {
                    deleteRelationship(rel);
                }
            }
        }

        public void deleteRelationship(Relationship relationship) {
            relationship.getStartNode().removeRelationship(relationship);
            relationship.getEndNode().removeRelationship(relationship);
            List<Relationship>                    relList                  =
relationships.get(relationship.getType());
            if (relList != null) {
                relList.remove(relationship);
            }
        }
    }
```

ДОДАТОК Б

Звіт з результатами перевірки на унікальність тексту в базі ХНУРЕ

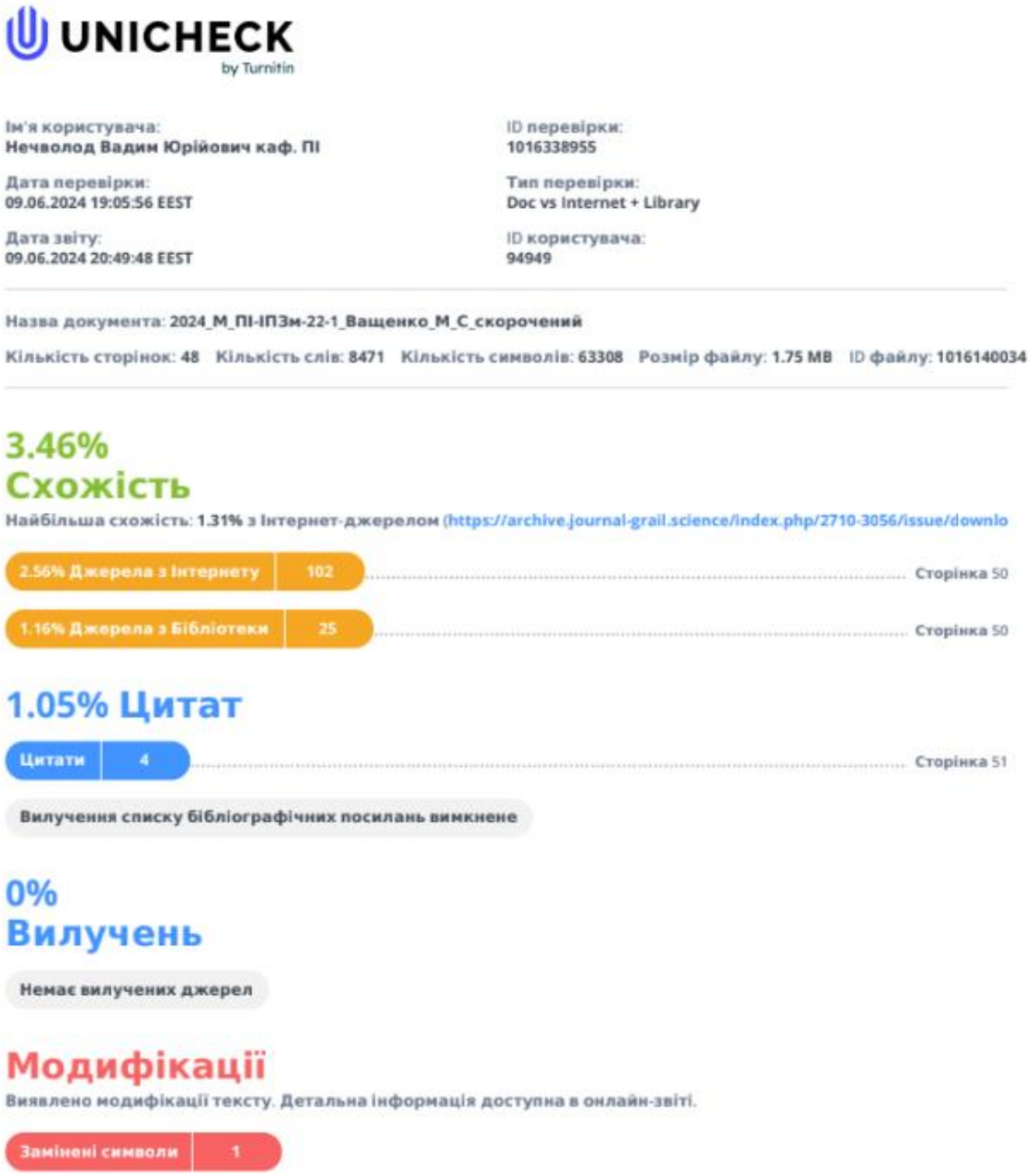


Рисунок Б.1 – Результат перевірки кваліфікаційної роботи магістра на плагіат

ДОДАТОК В

Слайди презентації

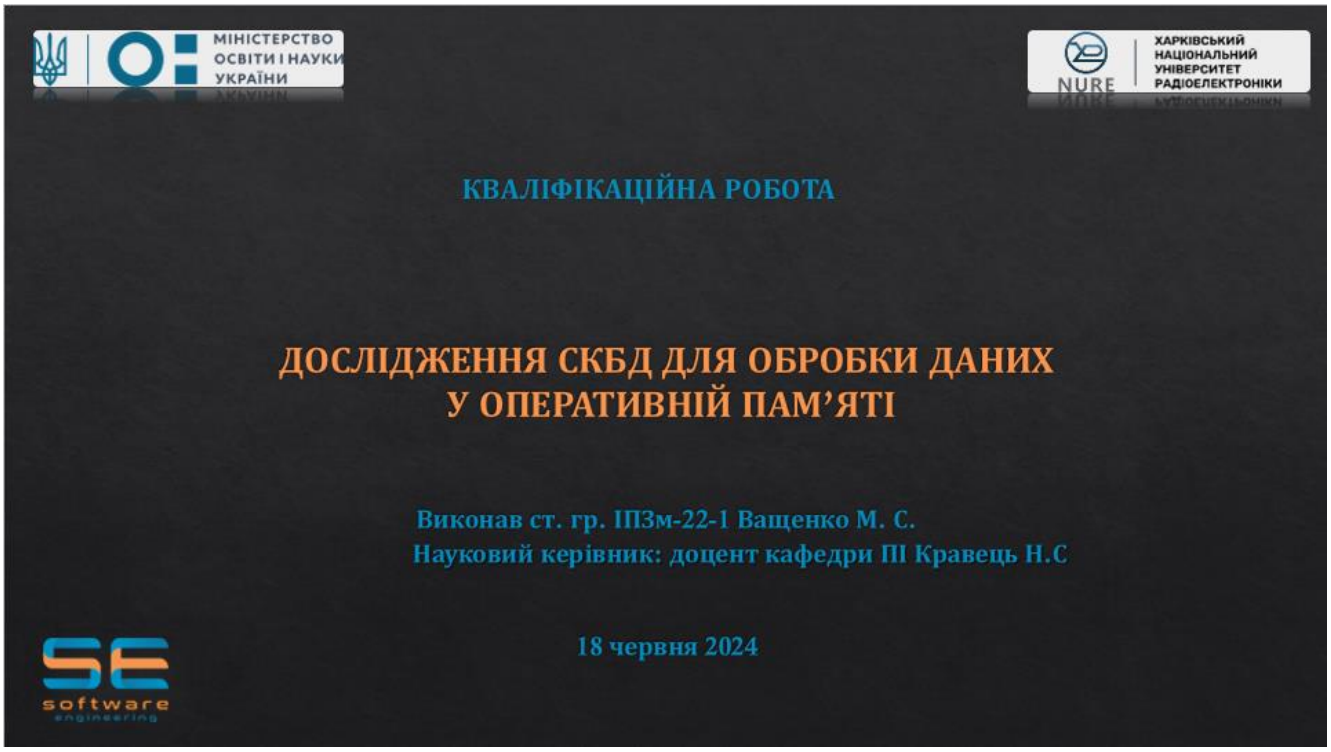


Рисунок В.1 – Слайд 1

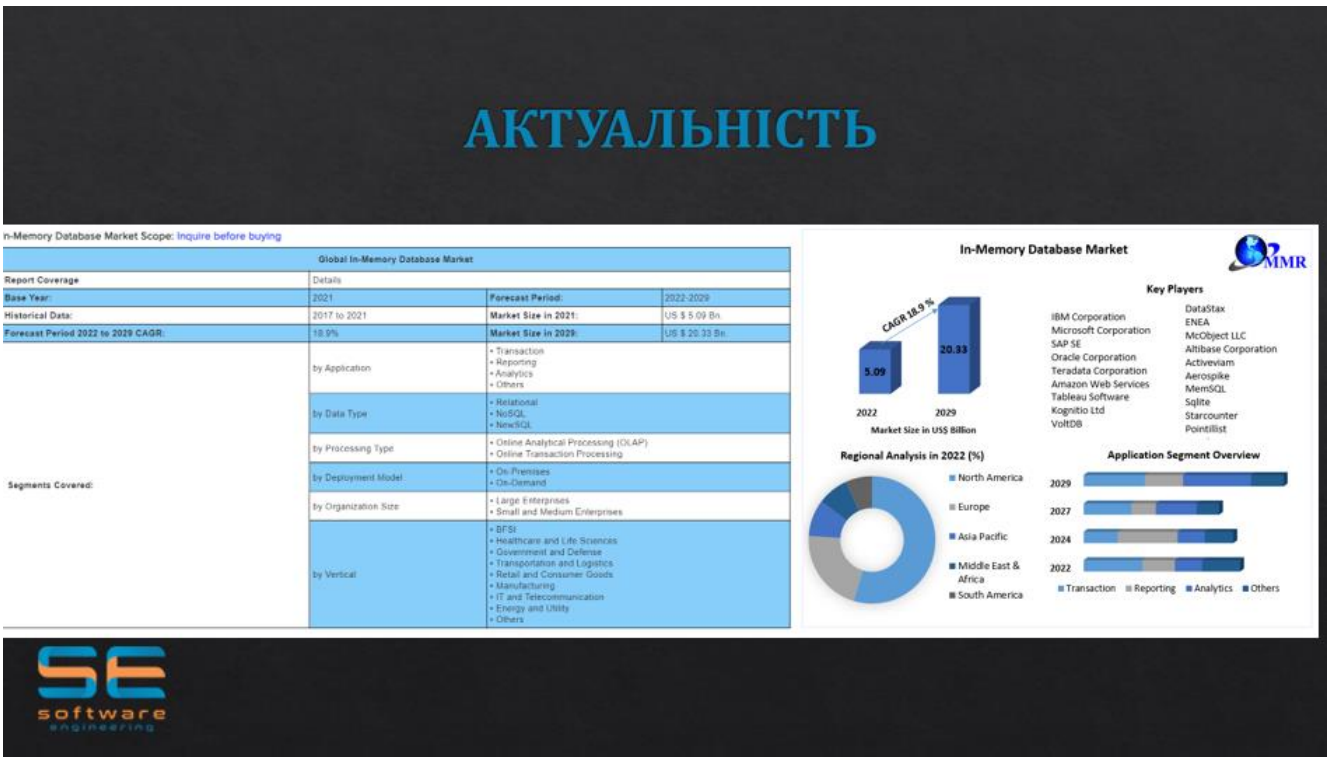


Рисунок В.2 – Слайд 2

Рисунок В.3 – Слайд 3

Рисунок В.4 – Слайд 4

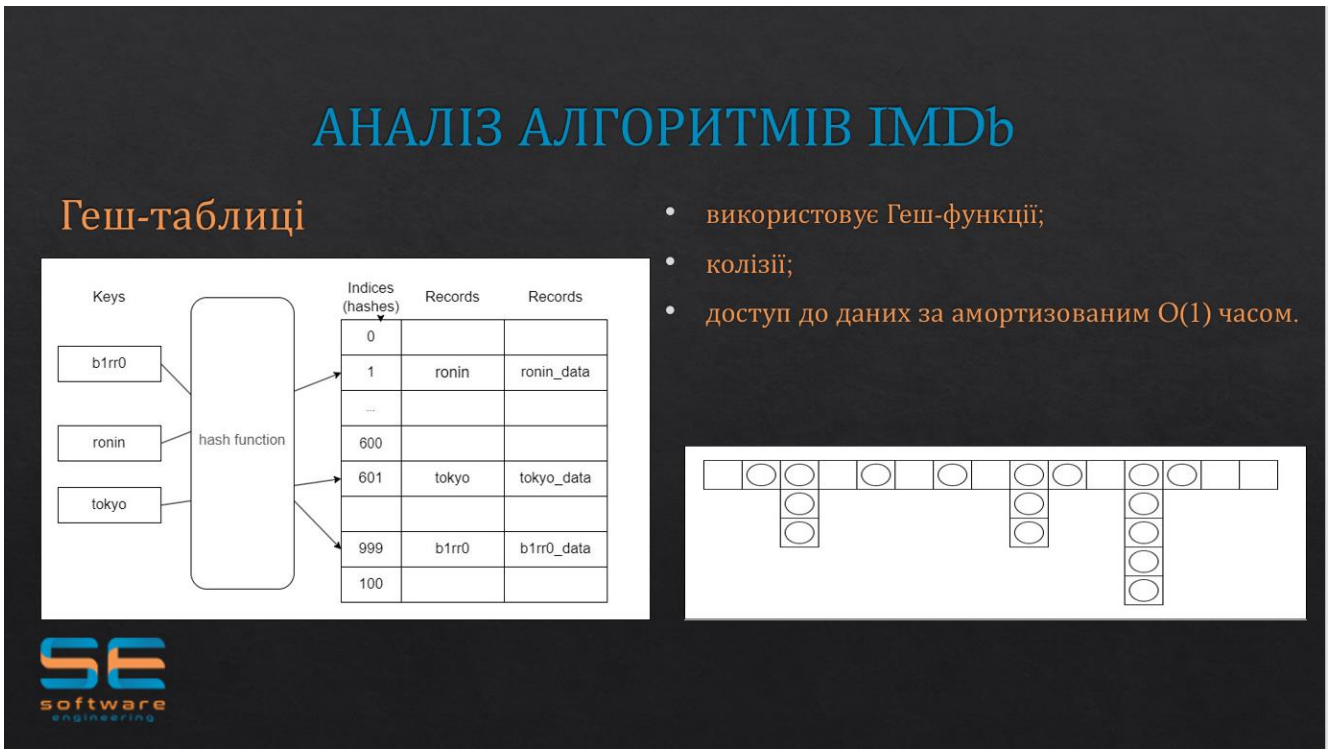


Рисунок В.5 – Слайд 5

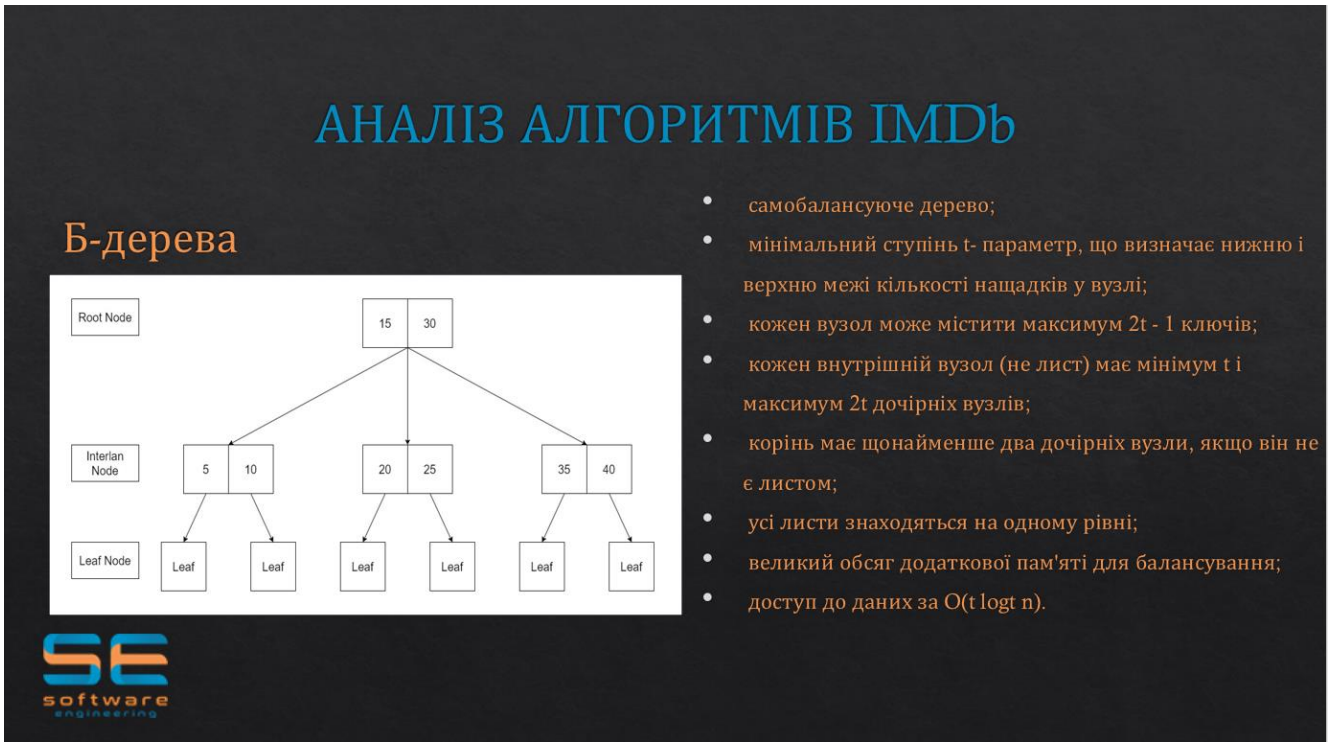


Рисунок В.6 – Слайд 6

Рисунок В.7 – Слайд 7

Рисунок В.8 – Слайд 8

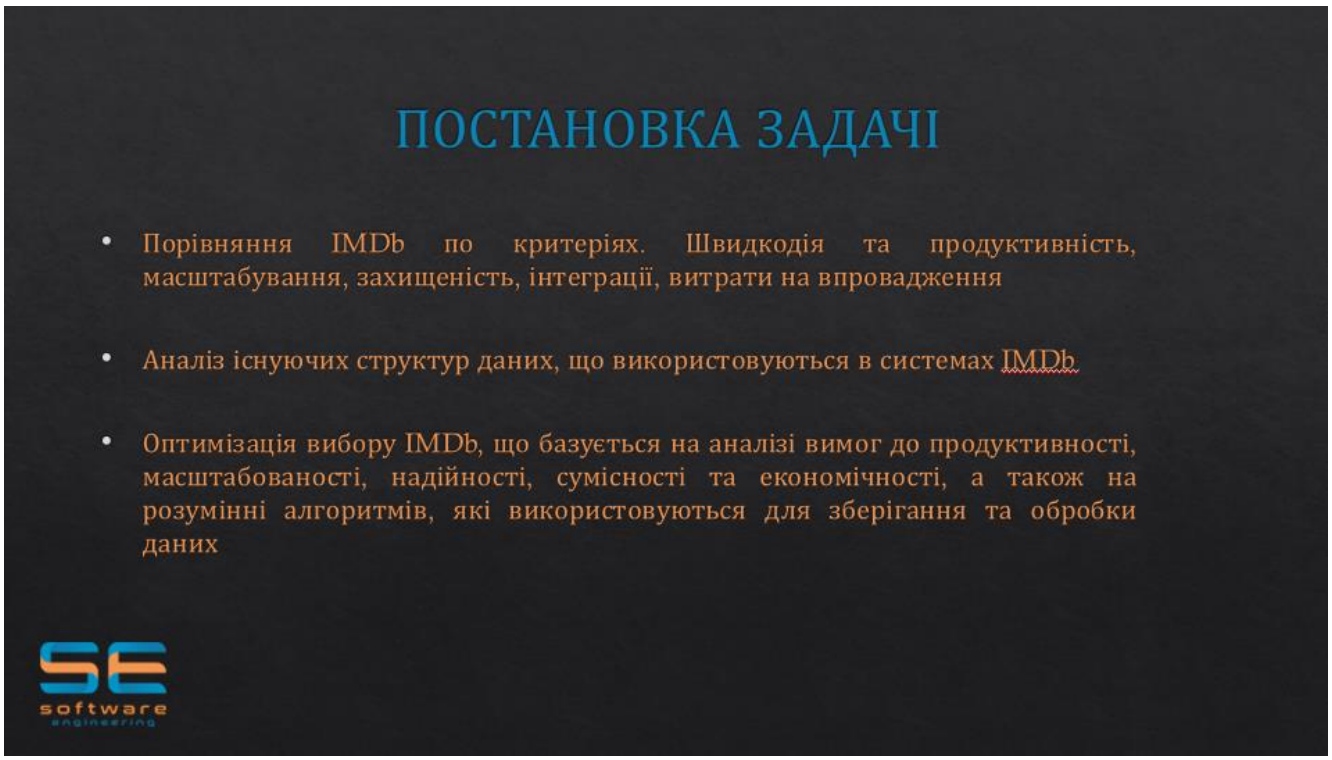


Рисунок В.9 – Слайд 9

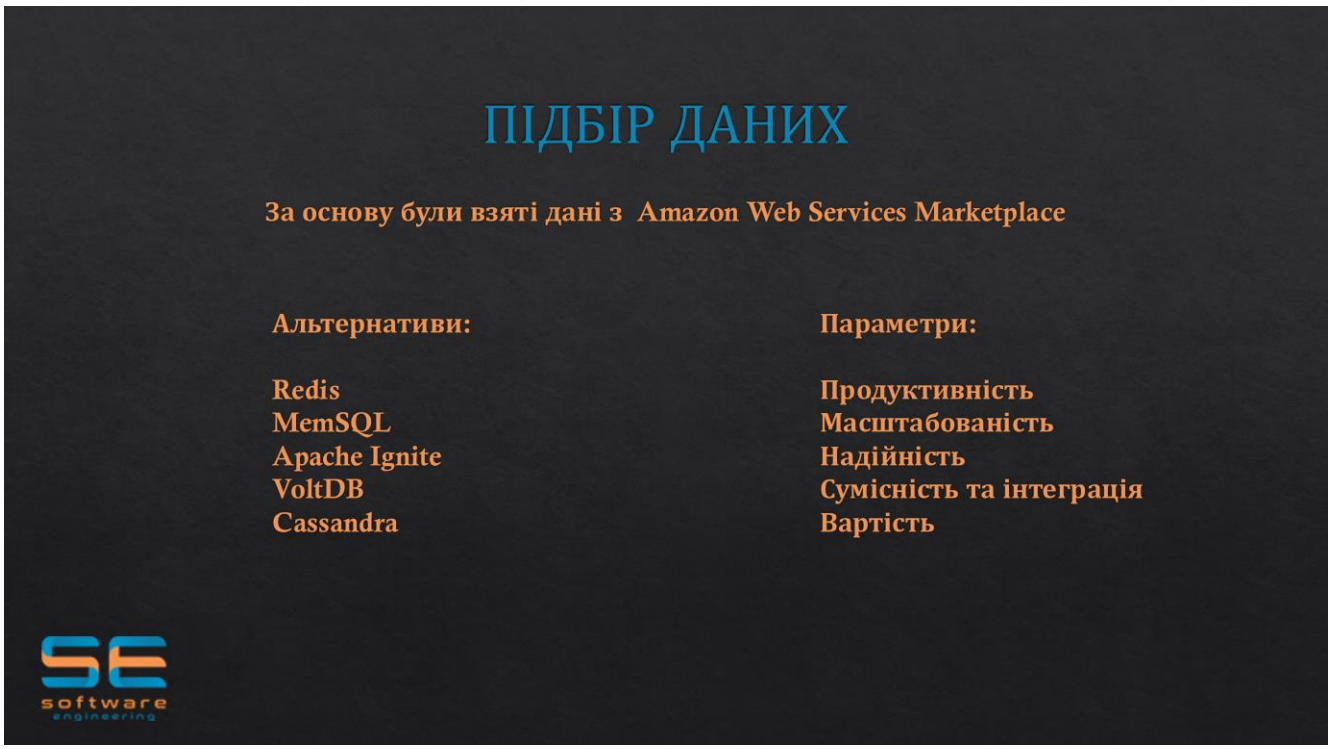


Рисунок В.10 – Слайд 10

Рисунок В.11 – Слайд 11

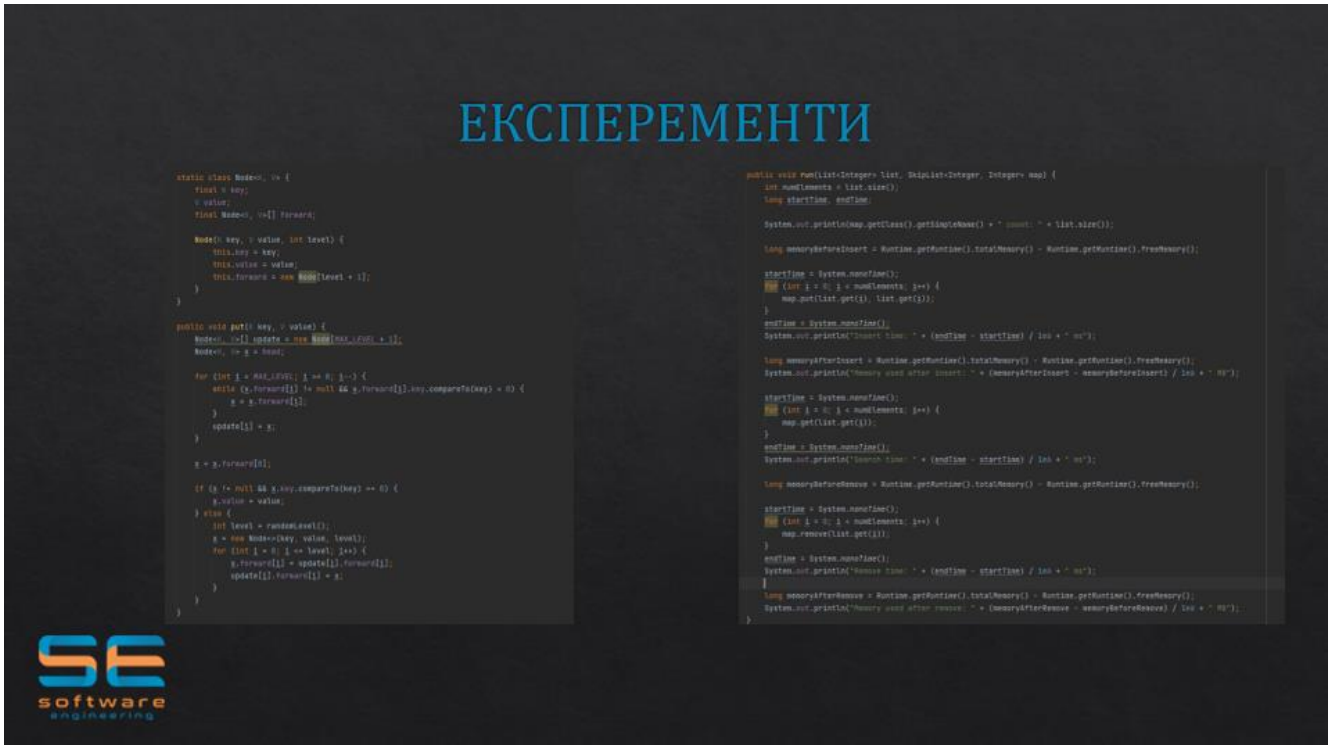


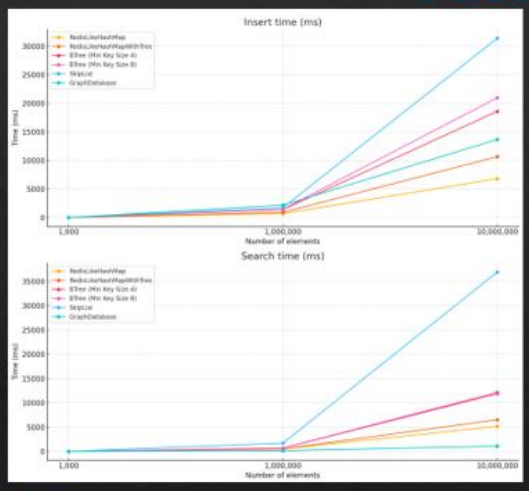
Рисунок В.12 – Слайд 12

РЕЗУЛЬТАТИ

- Хеш-таблиці з відкритою адресацією показали кращі результати по вставці та видаленню ключів для невеликих та середніх обсягів даних, але поступалися іншим структурам при пошуку.
- Хеш-таблиці з бакетами на основі бінарних дерев продемонстрували більш стабільний час пошуку, але вимагали більше пам'яті.
- Б-дерева забезпечили хорошу продуктивність при великих обсягах даних завдяки збалансованій структурі, але їхня реалізація виявилася складнішою.
- Скіп-списки показали найкращий час вставки для великих обсягів даних, але потребували більше пам'яті та мали гірший час видалення.
- Графові сховища даних були ефективними для складних зв'язків між даними, але їхня продуктивність знижувалася з ростом кількості зв'язків, також там зберігаються зв'язки поміж нодами, тому пам'яті використовується значно більше.

Рисунок В.13 – Слайд 13

РЕЗУЛЬТАТИ



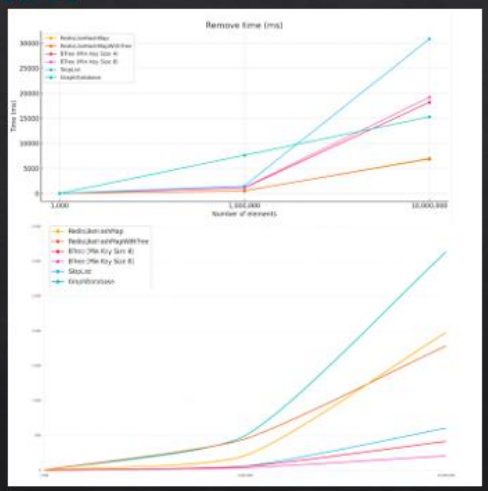


Рисунок В.14 – Слайд 14

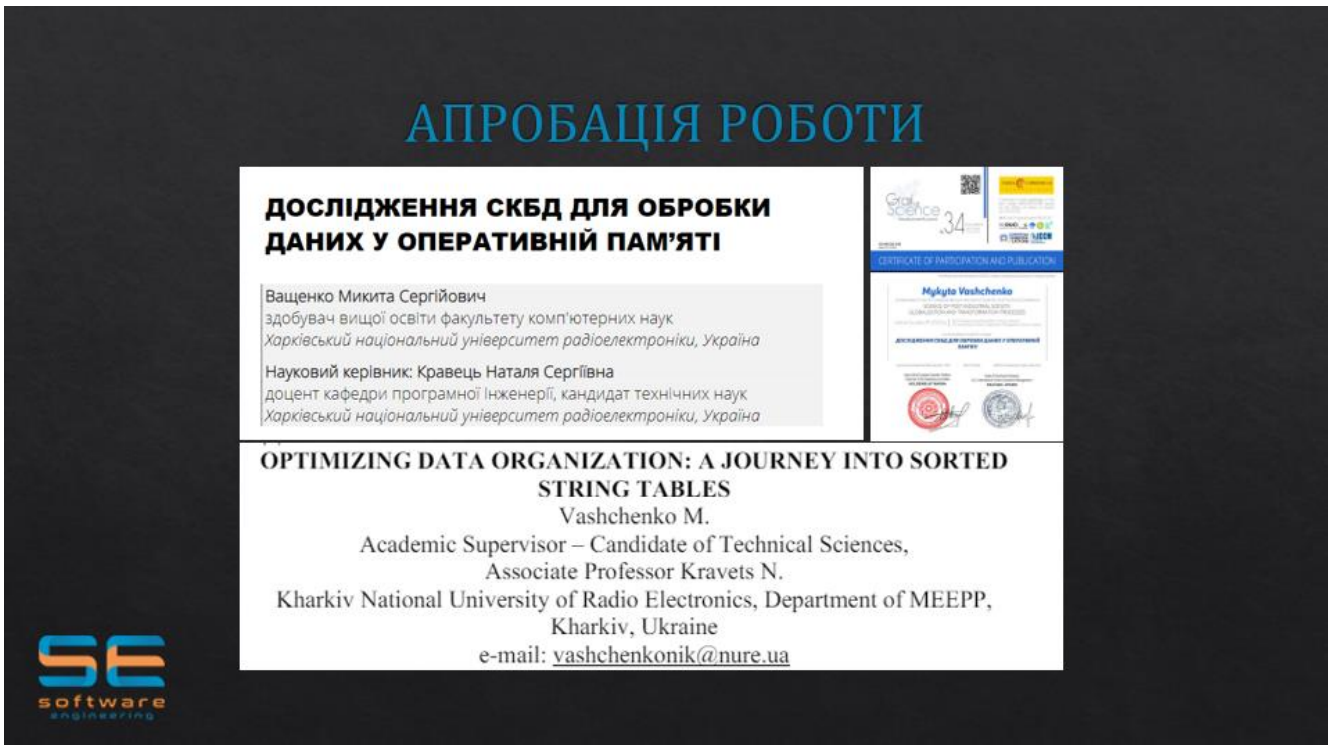


Рисунок В.15 – Слайд 15

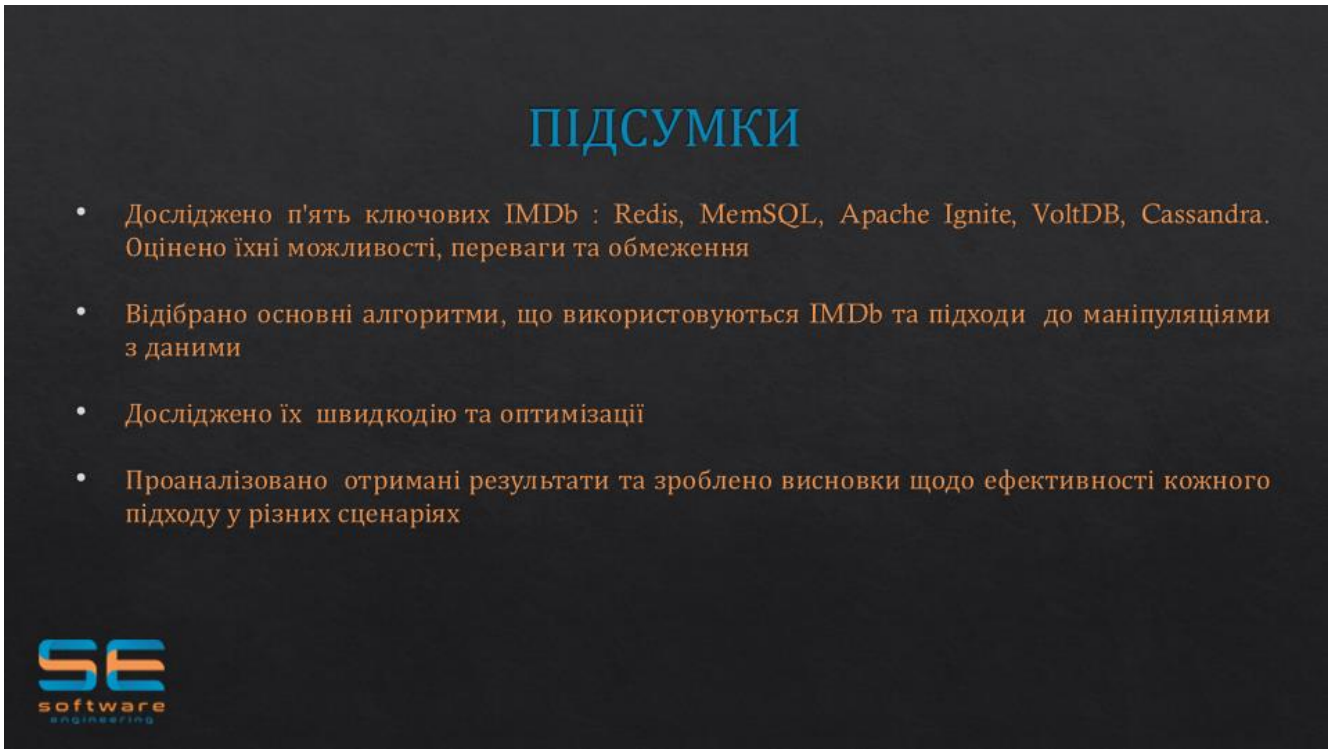


Рисунок В.16 – Слайд 16

ДОДАТОК Г

Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008:2015

Експертний висновок результатів перевірки кваліфікаційної роботи

студент (посада) | програмної інженерії (кафедра) | ІПЗм-22-1 (група)

Ващенко Микита Сергійович
(прізвище, ім'я, по батькові)

Зауваження

Пункт ДСТУ 3008-2015	Зміст пункту	Сторінка кваліфікаційної роботи
1	2	3
	7.1 Загальні положення	
	7.3 Нумерація сторінок звіту	
	7.4 Нумерація розділів, підрозділів, пунктів, підпунктів	
	7.5 Рисунки	
	7.6 Таблиці	
	7.7 Переліки	
	7.8 Примітки	
	7.9 Виноски	
	7.10 Формули та рівняння	
	7.11 Посилання	
	7.13 Список авторів	
	7.14 Скорочення та умовні позначки	
	7.15 Додатки	

зауважень немає

Експерт _____ (підпис) Олена ОЛІЙНИК (прізвище, ініціали)

11.06.2024

ДОДАТОК Д

Апробація кваліфікаційної роботи

International scientific journal «Grail of Science» | № 34 (December, 2023)

ABSTRACT

DOI 10.36074/grail-of-science.08.12.2023.37

ДОСЛІДЖЕННЯ СКБД ДЛЯ ОБРОБКИ ДАНИХ У ОПЕРАТИВНІЙ ПАМ'ЯТІ

Ващенко Микита Сергійович
здобувач вищої освіти факультету комп'ютерних наук
Харківський національний університет радіоелектроніки, Україна

Науковий керівник: Кравець Наталя Сергіївна
доцент кафедри програмної інженерії, кандидат технічних наук
Харківський національний університет радіоелектроніки, Україна

Мета цієї наукової публікації полягає в підтримці подальших досягнень у галузі систем керування базами даних для операційної обробки даних у оперативній пам'яті. Також вона спрямована на полегшення обміну знань та досвіду серед науковців і фахівців.

Почнемо з порівняння основних видів пам'яті, що випростовуються СКБД RAM (Random Access Memory) та SSD (Solid-state drive), які представлені у таблиці 1.

Таблиця 1

Порівняльна характеристика RAM та SSD

Характеристики	RAM	SSD
Швидкість доступу	Дуже висока, вимірюється в наносекундах	Низька відносно RAM, вимірюється в мікросекундах.
Швидкість передачі даних	Велика, виражається в гігабайтах за секунду.	Середня, виражається в мегабайтах за секунду.
Тип доступу	Прямий доступ, безперервний, використовується для зберігання тимчасових даних під час роботи програм	Зазвичай послідовний або випадковий доступ до даних, використовується для зберігання даних на постійному носії.
Втрати даних	Втрати при вимкненні живлення, тимчасове сховище	Зберігає дані після вимкнення живлення, постійне сховище
Використання	Використовується для тимчасового зберігання активних даних та запущених програм.	Використовується для постійного зберігання даних та програм.
Ціна	Висока	Відносно низька, на 2 порядки нижче за RAM

Для взаємодій з даними використовується обидва типи пам'яті, проте ми сфокусуємося на системах керування базами даних для роботи з даними у оперативній пам'яті [1].

Ці системи призначені для забезпечення швидкого та ефективного доступу до даних. Такий підхід особливо корисний для завдань, які вимагають

Рисунок Д.1 – Сторінка 1

високої швидкості обробки та низької затримки. Основні сценарії використання включають:

1. **Швидкий доступ до даних.** Доступ набагато швидше, порівняно з традиційними базами даних на дисках. Це дозволяє оптимізувати завдання, які вимагають миттєвої відповіді, такі як реальний час обслуговування запитів.

2. **Кешування.** Використання оперативної пам'яті для кешування прискорює доступ до часто використовуваних даних, що значно зменшує час відповіді на запити.

3. **Реальний час та потокова обробка.** Ці системи використовуються для обробки потокових даних у реальному часі, де низька затримка при доступі до даних має вирішальне значення.

4. **Висока паралельність.** Використання в пам'яті дозволяє досягати високого рівня паралельності, що сприяє ефективному використанню ресурсів системи.

Оптимізація алгоритмів та обчислень. Дані в пам'яті дозволяють використовувати оптимізовані алгоритми для прискорення обчислень [2].

Системи керування базами даних (СКБД) для роботи з даними у оперативній пам'яті можна класифікувати за різними критеріями, а саме:

1. In-Memory Key-Value Stores

Redis є надзвичайно швидкою базою даних у пам'яті, основаною на структурах даних типу ключ-значення. Підтримує ряд типів даних, включаючи строки, хеші, списки, множини та сортовані множини. Забезпечує можливість кешування та зберігання тимчасових даних для підвищення швидкодії додатків.

2. In-Memory Columnar Databases

SAP HANA використовує колонкове зберігання для високошвидкісної обробки аналітичних операцій в реальному часі. Дозволяє використовувати дані в пам'яті для прискорення операцій з даними та забезпечення великої швидкодії. Підтримує аналітичні можливості для бізнес-запитів та звітів.

Google Bigtable використовує модель зберігання даних у вигляді колонок та є розподіленою системою для масштабованої роботи з даними. Оптимізований для швидкого доступу до великих обсягів структурованих даних. Використовується Google для забезпечення великої доступності та високої швидкодії внутрішніх служб.

3. In-Memory Document Stores

Couchbase - це NoSQL база даних, яка використовується для зберігання даних у форматі документів, основаних на JSON. Забезпечує гнучкість у роботі з різними типами даних та швидкий доступ до документів у пам'яті. Використовується для різних завдань, включаючи розробку веб-додатків та аналіз даних.

4. In-Memory Analytical Databases

Apache Spark працює у пам'яті та дозволяє виконувати розподілені обчислення та аналіз даних у реальному часі. Забезпечує велику швидкодію завдяки утриманню даних у пам'яті під час обчислень. Використовується для обробки великих обсягів даних та машинного навчання.

5. In-Memory NewSQL Databases

VoltDB використовує архітектуру у пам'яті для забезпечення високої швидкодії та транзакційної стійкості даних. Оптимізований для роботи в

Рисунок Д.2 – Сторінка 2

реальному часі та обробки транзакцій. Використовується для вимогливих до швидкодії додатків, таких як фінансові системи та телекомунікаційні платформи.

NuoDB пропонує розподілену архітектуру, горизонтальну масштабованість та транзакційну стійкість. Дозволяє масштабувати систему вгору та вниз в залежності від навантаження. Використовується для великих та складних додатків, де потрібна стійкість та масштабованість.

6. In-Memory Graph Databases

Neo4j є графовою базою даних, яка ефективно зберігає та опрацьовує графові структури даних. Забезпечує гнучкість для моделювання та аналізу зв'язків між об'єктами. Використовується для сценаріїв, де важлива робота зі зв'язками, таких як соціальні мережі та аналіз мереж.

Робота з даними у оперативній пам'яті принесла значну швидкодію та реактивність систем, але вносить свої виклики. Обмеженість пам'яті може обмежувати розмір та тип даних, що зберігаються. Несталістичність даних може вести до втрати деякої інформації при збоях чи перезавантаженні. Синхронізація та консистентність між компонентами системи стають викликом при високих навантаженнях. За перевагами варто відзначити покращену швидкодію завдяки оперативній пам'яті, миттєву обробку запитів для високої реактивності та підтримку реального часу для систем, які працюють з даними у реальному часі, такими як фінансові операції чи моніторинг подій.

Дослідження в галузі СКБД для обробки даних у оперативній пам'яті є актуальним та перспективним. Подальший розвиток та оптимізація існуючих технологій можуть призвести до нових інновацій, спрямованих на вирішення викликів, пов'язаних із сталістичністю даних та обмеженістю пам'яті. Правильний вибір та ефективне використання СКБД можуть значно покращити продуктивність систем та забезпечити більш ефективну роботу додатків у реальному часі, враховуючи вказані виклики та переваги.

Список використаних джерел:

[1] Мартин Клеппман. (2018) Високонавантажених програми. Програмування, масштабування, підтримка Київ: "Глобал-Тренд", 202-255.

[2] Джошиан Карлсон. (2013) "Redis in Action" підтримка Київ: «O'Reilly Media», 152-245.

Рисунок Д.3 – Сторінка 3

УДК 004.6

OPTIMIZING DATA ORGANIZATION: A JOURNEY INTO SORTED STRING TABLES

Vashchenko M.
Academic Supervisor – Candidate of Technical Sciences,
Associate Professor Kravets N.
Kharkiv National University of Radio Electronics, Department of MEEPP,
Kharkiv, Ukraine
e-mail: vashchenkonik@nure.ua

This article delves into the adoption of Sorted String Tables (SSTables) as a solution for efficient and organized data storage in modern information systems. It addresses the challenges posed by disorderly data arrangement and presents SSTables as a remedy, with a focus on key aspects such as key uniqueness and the compaction process. The advantages of SSTables, including streamlined data merging and two-level indexing, are underscored alongside strategies for risk minimization. Moreover, the article highlights the widespread adoption of SSTables in various data stores and distributed systems, advocating for further research to optimize performance and enhance data recovery mechanisms.

In contemporary information systems, there is a pressing need for efficient and structured data storage. Disorderly arrangement of keys and their associated values in journal tables presents a significant challenge. This article discusses the shift towards a more organized data storage approach using SSTables, along with methods to ensure data orderliness on disk.

Ensuring key uniqueness in each data segment is vital for SSTables' effective operation, aiming to prevent data duplication and maintain information consistency. The compaction process plays a pivotal role in achieving this goal by merging and restructuring data segments, eliminating duplication, and ensuring key uniqueness. This involves analyzing, ordering, and merging keys from different segments to meet uniqueness requirements, thereby preserving data order and integrity in SSTables. Additionally, data management mechanisms like replication strategies, data version control, and conflict resolution further contribute to ensuring key uniqueness, reinforcing the structuredness and integrity of data in SSTables.

The advantages of transitioning to SSTables encompass a wide array of benefits that significantly enhance the efficiency and performance of data storage systems. One notable advantage is the capability to merge data, even when the volume of keys exceeds available RAM. This capability opens up new possibilities for handling large volumes of data, previously hindered by memory limitations. Transitioning to SSTables, based on the merge sort algorithm, offers an efficient mechanism for combining data from different segments, enabling optimal utilization of available resources. Moreover, data merging facilitates

Рисунок Д.4 – Сторінка 1

effective management of changing data volumes and enables dynamic scaling of system performance according to application requirements. Furthermore, the data merging mechanism ensures proper handling of matching keys, retaining only the latest values for identical keys, thereby reducing information duplication and optimizing data storage utilization. Consequently, transitioning to SSTables not only ensures efficient management of data volumes but also guarantees the integrity and consistency of information in storage.

Data block compression is essential for efficient utilization of storage resources and bandwidth in data storage systems. Maintaining data orderliness in SSTables necessitates the application of a two-level indexing strategy. This strategy involves sorting data both on disk and in memory to enhance query performance and ensure rapid access to data. Data indexing occurs on disk at the first level, using specialized data structures such as B-trees or LSM-trees, facilitating quick data retrieval and efficient disk space utilization. The second level of indexing, conducted in memory using data structures such as hash tables or search trees like AVL or red-black trees, enables fast access to data in memory, minimizing system response time to queries. Thus, the two-level indexing strategy, both in memory and on disk, ensures efficient data management in SSTables, facilitating swift and convenient access to information.

The transition to SSTables not only streamlines data storage but also facilitates the formation of optimal queries by ensuring data orderliness and efficiency. The systematic arrangement of data segments in SSTables enables streamlined query execution, reducing computational overhead and latency. Additionally, the inherent compaction process aids in maintaining query integrity by eliminating redundant data and ensuring key uniqueness, thereby enhancing the accuracy and reliability of query result [1].

The vulnerability of data storage systems based on memtable, particularly in the event of application failures, poses a risk of data loss. To mitigate this risk, a combined mechanism for saving data to disk and memory is proposed. Memtable is utilized for quick data recording to RAM before their long-term storage on disk in SSTables. Periodic data recording to disk ensures reliable data recovery following application failures, minimizing the risk of data loss and ensuring data integrity in storage.

References:

1. Shubin, I., & Kozyriev, A. (2023). Method for solving quantifier linear equations for formation of optimal queries to databases. Proceedings of the 7th International Conference on Computational Linguistics and Intelligent Systems (COLINS-2023). CEUR Workshop Proceedings, 3396, 449–459. https://ceur-ws.org/Vol-3396/paper36.pdf.

Рисунок Д.5 – Сторінка 2