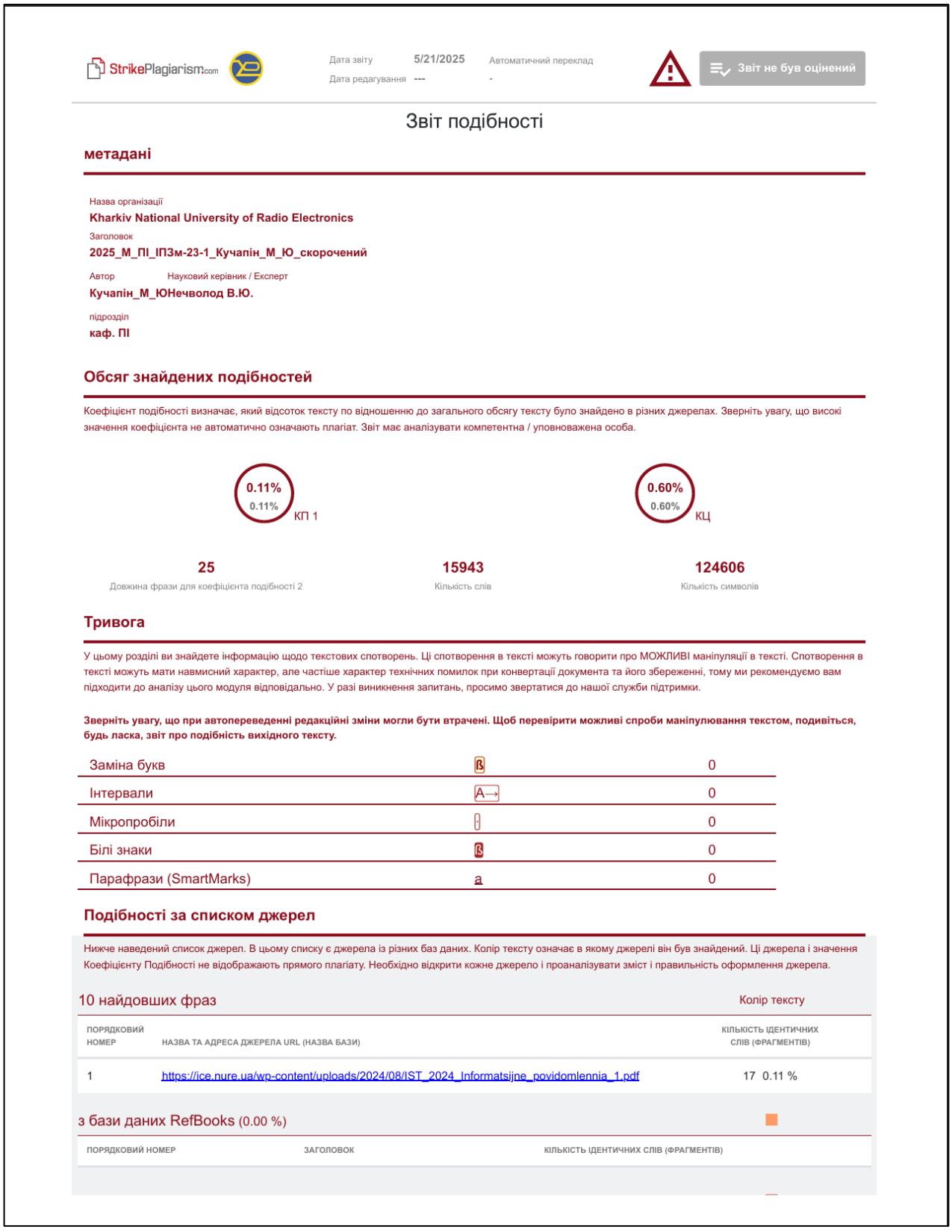
ДОДАТОК А

Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ



StrikePlagiarism.com

Дата звіту 5/21/2025 Автоматичний переклад

Дата редагування --- -

Звіт не був оцінений

Звіт подібності

метадані

Назва організації
Kharkiv National University of Radio Electronics
Заголовок
2025_М_ПІ_ІПЗм-23-1_Кучапін_М_Ю_скорочений
Автор Науковий керівник / Експерт
Кучапін_М_ЮНечволод В.Ю.
підрозділ
каф. ПІ

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

0.11% 0.11% КП 1

0.60% 0.60% КЦ

25 Довжина фрази для коефіцієнта подібності 2

15943 Кількість слів

124606 Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Зверніть увагу, що при автопереведенні редакційні зміни могли бути втрачені. Щоб перевірити можливі спроби маніпулювання текстом, подивіться, будь ласка, звіт про подібність вихідного тексту.

Заміна букв		0
Інтервали		0
Мікропробіли		0
Білі знаки		0
Парафрази (SmartMarks)	a	0

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://ice.nure.ua/wp-content/uploads/2024/08/IST_2024_Informatsijne_povidomlennia_1.pdf	17 0.11 %

з бази даних RefBooks (0.00 %)

ПОРЯДКОВИЙ НОМЕР	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)

Рисунок А.1 – Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ

(рисунок виконаний самостійно)

ДОДАТОК Б

Код адаптивного кеш клієнта

```
using RedisCacheOptimizer.Configurations;
using RedisCacheOptimizer.Interfaces;
using RedisCacheOptimizer.Utilities;
using StackExchange.Redis;

namespace RedisCacheOptimizer;

public class AdaptiveCacheClient : BaseDatabaseClient,
IAdaptiveCacheClient
{
    private readonly AdaptiveCacheClientOptions _options;
    private CancellationTokenSource _cancellationTokenSource = new();
    public AdaptiveCacheManager CacheManager;
    private readonly SemaphoreSlim _cleanupSemaphore = new
SemaphoreSlim(1, 1);

    public int ExpiredKeys
    {
        get
        {
            return _expiredKeys;
        }
    }

    private int _expiredKeys;

    public AdaptiveCacheClient(string connectionString,
AdaptiveCacheClientOptions options) : base(connectionString)
    {
        options.Validate();

        _options = options;

        CacheManager = new AdaptiveCacheManager(connectionString,
options.AdaptiveCacheManagerOptions);

        StartBackgroundCleanup();
    }

    public async Task StoreObjectAsync<T>(string key, T obj)
    {
        await HandleInsertWithMemoryManagementAsync(key, obj, async
() =>
        {
            var hashEntries = ObjectSerializer.FlattenObject(obj)
                .Select(kvp => new HashEntry(kvp.Key, kvp.Value))
                .ToArray();

            await _database.HashSetAsync(key, hashEntries);
            await UpdateKeyAccessAsync(key, false);
        });
    }
```

```
        public async Task<T> GetObjectAsync<T>(string key) where T :
new()
        {
            var hashEntries = await _database.HashGetAllAsync(key);
            if (hashEntries.Length == 0)
                return default;

            var properties = hashEntries.ToDictionary(entry =>
(string)entry.Name, entry => (string)entry.Value);

            var obj = ObjectSerializer.UnflattenObject<T>(properties);

            await UpdateKeyAccessAsync(key, true);

            return obj;
        }

        public async Task SetKeyTTLAsync(string key, TimeSpan ttl)
        {
            await SetPropertyTTLAsync(key, string.Empty, ttl);
        }

        public async Task SetPropertyTTLAsync(string key, string
propertyPath, TimeSpan ttl)
        {
            var sortedSetKey = key + Constants.TtlSortedSetSuffix;
            var now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();
            var expiryTime = now + (long)ttl.TotalSeconds;

            await _database.SortedSetAddAsync(sortedSetKey, propertyPath,
expiryTime);

            Log($"TTL for property '{propertyPath}' is set to
{ttl.TotalSeconds} seconds.");
        }

        public async Task<RedisValue> HashGetAsync(RedisKey key,
RedisValue field)
        {
            var value = await _database.HashGetAsync(key, field);
            if (!value.IsNull)
            {
                await UpdatePropertyAccessAsync(key, field);
            }

            return value;
        }

        public async Task<bool> HashSetAsync(RedisKey key, RedisValue
field, RedisValue value)
        {
            var combinedObject = new { Field = field.ToString(), Value =
value.ToString() };

            bool operationResult = false;

            await HandleInsertWithMemoryManagementAsync(key,
combinedObject, async () =>
```

```
            {
                operationResult = await _database.HashSetAsync(key,
field, value);

                await UpdatePropertyAccessAsync(key, field);
            });

            return operationResult;
        }

        public async Task<bool> SetAddAsync(RedisKey key, RedisValue
value)
        {
            bool operationResult = false;

            await HandleInsertWithMemoryManagementAsync(key,
value.ToString(), async () =>
            {
                operationResult = await _database.SetAddAsync(key,
value);

                if (operationResult)
                {
                    await UpdatePropertyAccessAsync(key, value);
                }
            });

            return operationResult;
        }

        public async Task<bool> SetContainsAsync(RedisKey key, RedisValue
value)
        {
            var exists = await _database.SetContainsAsync(key, value);

            if (exists)
            {
                await UpdatePropertyAccessAsync(key, value);
            }

            return exists;
        }

        public async Task<long> ListLeftPushAsync(RedisKey key,
RedisValue value)
        {
            long result = 0;

            await HandleInsertWithMemoryManagementAsync(key,
value.ToString(), async () =>
            {
                result = await _database.ListLeftPushAsync(key, value);
                await UpdatePropertyAccessAsync(key, value);
            });

            return result;
        }
```

```
        public async Task<RedisValue> ListLeftPopAsync(RedisKey key)
        {
            var value = await _database.ListLeftPopAsync(key);
            if (!value.IsNull)
            {
                await RemoveElementMetadataAsync(key, value);
            }
            return value;
        }

        public async Task<RedisValue[]> ListRangeAsync(RedisKey key, long
start = 0, long stop = -1)
        {
            var values = await _database.ListRangeAsync(key, start,
stop);
            foreach (var value in values)
            {
                await UpdatePropertyAccessAsync(key, value);
            }
            return values;
        }

        public async Task<bool> SortedSetAddAsync(RedisKey key,
RedisValue member, double score)
        {
            var combinedObject = new { Member = member.ToString(), Score
= score };
            bool operationResult = false;

            await HandleInsertWithMemoryManagementAsync(key,
combinedObject, async () =>
            {
                operationResult = await _database.SortedSetAddAsync(key,
member, score);

                if (operationResult)
                {
                    await UpdatePropertyAccessAsync(key, member);
                }
            });

            return operationResult;
        }

        public async Task<RedisValue[]>
SortedSetRangeByRankAsync(RedisKey key, long start = 0, long stop = -1)
        {
            var values = await _database.SortedSetRangeByRankAsync(key,
start, stop);
            foreach (var value in values)
            {
                await UpdatePropertyAccessAsync(key, value);
            }
            return values;
        }

        public async Task<TimeSpan?> GetKeyTTLAsync(string key)
        {
```

```
            return await GetPropertyTTLAsync(key, string.Empty);
        }

        public async Task<TimeSpan?> GetPropertyTTLAsync(string key,
string propertyPath)
        {
            var sortedSetKey = key + Constants.TtlSortedSetSuffix;
            var expiryTimestamp = await
_database.SortedSetScoreAsync(sortedSetKey, propertyPath);

            if (expiryTimestamp.HasValue)
            {
                var now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();
                var ttlSeconds = expiryTimestamp.Value - now;

                if (ttlSeconds > 0)
                {
                    return TimeSpan.FromSeconds(ttlSeconds);
                }
                else
                {
                    return TimeSpan.Zero;
                }
            }

            return null;
        }

        public async Task EnableProtectionAsync(string key, string
propertyPath = null)
        {
            await SetProtectionAsync(key, propertyPath, true);
        }

        public async Task DisableProtectionAsync(string key, string
propertyPath = null)
        {
            await SetProtectionAsync(key, propertyPath, false);
        }

        public void Dispose()
        {
            StopBackgroundCleanup();
            _redis?.Dispose();
        }

        private async Task
HandleInsertWithMemoryManagementAsync<T>(string key, T value, Func<Task>
insertAction)
        {
            long estimatedSize =
SizeEstimator.CalculateObjectSizeInBytes(value);
            long adjustedSize = (long)(estimatedSize * 1.5);

            await
CacheManager.CleanupForRequiredMemoryAsync(adjustedSize);

            await insertAction();
```

```
        }

        private async Task SetProtectionAsync(string key, string
propertyPath, bool enable)
        {
            if (string.IsNullOrEmpty(propertyPath))
            {
                if (enable)
                {
                    await _database.StringSetAsync(key +
Constants.ProtectedSuffix, "true");
                    Log($"Protection enabled for key '{key}'.");
                }
                else
                {
                    await _database.KeyDeleteAsync(key +
Constants.ProtectedSuffix);
                    Log($"Protection disabled for key '{key}'.");
                }
            }
            else
            {
                var metaKey = key + Constants.TtlHashSuffix;
                var field = propertyPath + ":protected";
                if (enable)
                {
                    await _database.HashSetAsync(metaKey, field, "true");
                    Log($"Protection enabled for property
'{propertyPath}' in key '{key}'.");
                }
                else
                {
                    await _database.HashDeleteAsync(metaKey, field);
                    Log($"Protection disabled for property
'{propertyPath}' in key '{key}'.");
                }
            }
        }

        private async Task UpdateKeyAccessAsync(string key, bool
recalculateTtl)
        {
            var metaKey = key + Constants.TtlHashSuffix;
            var frequency = await UpdateAccessStatsAsync(metaKey, "key");

            await CacheManager.UpdateEvictionScoreAsync(key);

            var lastAccessValue = await _database.HashGetAsync(metaKey,
"lastAccess");
            var shouldBeUpdated = true;

            if (recalculateTtl && shouldBeUpdated)
            {
                var sortedSetKey = key + Constants.TtlSortedSetSuffix;
                var existingTTL = await
_database.SortedSetScoreAsync(sortedSetKey, string.Empty);

                if (existingTTL.HasValue)
```

```
                {
                    TimeSpan ttl = await ComputeAdaptiveTTLAsync(metaKey,
"key");

                    var now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();
                    var expiryTime = now + (long)ttl.TotalSeconds;

                    await _database.SortedSetAddAsync(sortedSetKey,
string.Empty, expiryTime);
                    Log($"Adaptive TTL for key '{key}' recalculated:
{ttl.TotalSeconds} seconds.");
                }
                else
                {
                    Log($"Key '{key}' does not have a TTL set. Skipping
adaptive TTL.");
                }
            }
        }

        private async Task UpdatePropertyAccessAsync(string key, string
propertyPath)
        {
            var metaKey = key + Constants.TtlHashSuffix;

            await UpdateKeyAccessAsync(key, true);

            var frequency = await UpdateAccessStatsAsync(metaKey,
propertyPath);

            var lastAccessValue = await _database.HashGetAsync(metaKey,
$"{propertyPath}:lastAccess");
            var shouldBeUpdated = lastAccessValue.IsNullOrEmpty ||
DateTimeOffset.UtcNow.ToUnixTimeSeconds() - (long)lastAccessValue >
Constants.MIN_METADATA_UPDATE_INTERVAL;

            if (shouldBeUpdated)
            {
                var sortedSetKey = key + Constants.TtlSortedSetSuffix;
                var existingTTL = await
_database.SortedSetScoreAsync(sortedSetKey, propertyPath);

                if (existingTTL.HasValue)
                {
                    var now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();

                    TimeSpan ttl = await ComputeAdaptiveTTLAsync(metaKey,
propertyPath);
                    var expiryTime = now + (long)ttl.TotalSeconds;

                    await _database.SortedSetAddAsync(sortedSetKey,
propertyPath, expiryTime);
                    Log($"Adaptive TTL for property '{propertyPath}'
recalculated: {ttl.TotalSeconds} seconds.");
                }
                else
                {
```

```
                    Log($"Property '{propertyPath}' does not have a TTL
set. Skipping adaptive TTL.");
                }
            }
        }

        private async Task<double> UpdateAccessStatsAsync(string metaKey,
string accessKey)
        {
            var now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();

            var dataKey = metaKey.Replace(Constants.TtlHashSuffix, "");
            string prefix = accessKey == "key" ? "" : accessKey + ":";
            string totalAccessesField = $"{prefix}totalAccesses";
            string firstAccessTimeField = $"{prefix}firstAccessTime";
            string lastAccessField = $"{prefix}lastAccess";
            string frequencyField = $"{prefix}frequency";

            var firstAccessTimeValue = await
_database.HashGetAsync(metaKey, firstAccessTimeField);

            long totalAccesses = await
_database.HashIncrementAsync(metaKey, totalAccessesField, 1);
            long firstAccessTime = firstAccessTimeValue.IsNullOrEmpty ?
now : (long)firstAccessTimeValue;

            await _database.HashSetAsync(metaKey, new HashEntry[]
            {
                new HashEntry(totalAccessesField, totalAccesses),
                new HashEntry(firstAccessTimeField, firstAccessTime),
                new HashEntry(lastAccessField, now)
            });

            double elapsedTime = now - firstAccessTime;
            double frequency = elapsedTime > 0 ? totalAccesses /
elapsedTime : totalAccesses;

            await _database.HashSetAsync(metaKey, frequencyField,
frequency);

            string totalAccessSetMember = (accessKey == "key")
                ? dataKey
                : $"{dataKey}::{accessKey}";

            await
_database.SortedSetAddAsync(Constants.GlobalFrequencySetKey,
totalAccessSetMember, totalAccesses);
            await
_database.SortedSetAddAsync(Constants.GlobalLastAccessSetKey,
totalAccessSetMember, now);


            return frequency;
        }

        private async Task<TimeSpan> ComputeAdaptiveTTLAsync(string
metaKey, string accessKey)
        {
```

```
            string prefix = (accessKey == "key") ? "" : (accessKey +
":");
            string totalAccessesField = $"{prefix}totalAccesses";
            string lastAccessField = $"{prefix}lastAccess";

            var totalAccessesValue = await
_database.HashGetAsync(metaKey, totalAccessesField);
            var lastAccessValue = await _database.HashGetAsync(metaKey,
lastAccessField);

            long f = totalAccessesValue.IsNullOrEmpty ? 0 :
(long)totalAccessesValue;
            long tLastAccess = lastAccessValue.IsNullOrEmpty
                ? DateTimeOffset.UtcNow.ToUnixTimeSeconds()
                : (long)lastAccessValue;

            double now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();
            double f_decayed = f * Math.Exp(-_options.DecayLambda * (now
- tLastAccess));

            double maxTotalAccesses = 0;
            var totalAccessSetRange = await
_database.SortedSetRangeByRankWithScoresAsync(
                Constants.GlobalFrequencySetKey,
                0,
                0,
                Order.Descending);

            if (totalAccessSetRange != null && totalAccessSetRange.Length
> 0)
            {
                maxTotalAccesses = totalAccessSetRange[0].Score;
            }

            double f_normalized = (maxTotalAccesses > 0)
                ? Math.Min(f_decayed / maxTotalAccesses, 1.0)
                : 0.0;

            var ttlSec = _options.BaseTTL + (_options.MaxTTL -
_options.BaseTTL) * f_normalized;
            return ttlSec;
        }

        private void StopBackgroundCleanup()
        {
            _cancellationTokenSource.Cancel();
        }

        private void StartBackgroundCleanup()
        {
            Task.Run(async () =>
            {
                while
(!_cancellationTokenSource.Token.IsCancellationRequested)
                {
                    try
                    {
                        await CleanupExpiredPropertiesAsync();
```

```
                        await Task.Delay(_options.CleanupInterval,
_cancellationTokenSource.Token);
                    }
                    catch (TaskCanceledException)
                    {
                    }
                    catch (Exception ex)
                    {
                        Log($"Error during CleanupExpiredProperties:
{ex.Message}");
                    }
                }
            }, _cancellationTokenSource.Token);
        }

        private async Task CleanupExpiredPropertiesAsync()
        {
            await _cleanupSemaphore.WaitAsync();
            try
            {
                var server = _redis.GetServer(_redis.GetEndPoints()[0]);
                var keys = server.Keys(pattern: "*" +
Constants.TtlSortedSetSuffix);

                var now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();

                foreach (var key in keys)
                {
                    var dataKey =
key.ToString().Replace(Constants.TtlSortedSetSuffix, "");
                    var expiredProperties = await
_database.SortedSetRangeByScoreAsync(key, 0, now);

                    if (expiredProperties.Length > 0)
                    {
                        foreach (var propertyPath in expiredProperties)
                        {
                            Log($"The '{propertyPath}' property for the
key '{dataKey}' was deleted after the TTL expired.");

                            await
RemovePropertyAndSubPropertiesAsync(dataKey, key, propertyPath);
                            Interlocked.Increment(ref _expiredKeys);
                        }
                    }
                }
            }
            catch (Exception ex)
            {
                Log("Error during CleanupExpiredProperties: " +
ex.Message);
            }
            finally
            {
                _cleanupSemaphore.Release();
            }
        }
```

```
        private async Task RemovePropertyAndSubPropertiesAsync(string
dataKey, RedisKey ttlSortedSetKey, RedisValue propertyPath)
        {
            var hashFields = await _database.HashKeysAsync(dataKey);

            var fieldsToDelete = hashFields.Where(field =>
field.ToString().StartsWith(propertyPath.ToString())).ToArray();

            if (fieldsToDelete.Length > 0)
            {
                await _database.HashDeleteAsync(dataKey, fieldsToDelete);

                var metaKey = dataKey + Constants.TtlHashSuffix;
                var metaFieldsToDelete = fieldsToDelete.SelectMany(field
=> new RedisValue[]
                {
                    $"{field}:frequency",
                    $"{field}:lastAccess",
                    $"{field}:lastFrequencyUpdateTime"
                }).ToArray();

                await _database.HashDeleteAsync(metaKey,
metaFieldsToDelete);
                await _database.SortedSetRemoveAsync(ttlSortedSetKey,
fieldsToDelete);

                var members = fieldsToDelete.Select(field => new
RedisValue($"{dataKey}::{field}")).ToArray();
                await
_database.SortedSetRemoveAsync(Constants.GlobalFrequencySetKey, members);
                await
_database.SortedSetRemoveAsync(Constants.GlobalLastAccessSetKey, members);
                await
_database.SortedSetRemoveAsync(Constants.GlobalEvictionSetKey, members);
            }
            else
            {
                await _database.HashDeleteAsync(dataKey, propertyPath);
                await _database.SortedSetRemoveAsync(ttlSortedSetKey,
propertyPath);

                var metaKey = dataKey + Constants.TtlHashSuffix;
                await _database.HashDeleteAsync(metaKey, new RedisValue[]
                {
                    $"{propertyPath}:frequency",
                    $"{propertyPath}:lastAccess",
                    $"{propertyPath}:lastFrequencyUpdateTime"
                });

                var members = fieldsToDelete.Select(field => new
RedisValue($"{dataKey}::{field}")).ToArray();

                if (members.Length > 0)
                {
                    await
_database.SortedSetRemoveAsync(Constants.GlobalFrequencySetKey, members);
                    await
_database.SortedSetRemoveAsync(Constants.GlobalLastAccessSetKey, members);
```

```
                    await
_database.SortedSetRemoveAsync(Constants.GlobalEvictionSetKey, members);
                }

                string member = string.IsNullOrEmpty(propertyPath) ?
dataKey : $"{dataKey}::{propertyPath}";
                await
_database.SortedSetRemoveAsync(Constants.GlobalFrequencySetKey, member);
                await
_database.SortedSetRemoveAsync(Constants.GlobalLastAccessSetKey, member);
                await
_database.SortedSetRemoveAsync(Constants.GlobalEvictionSetKey, member);
            }
        }

        private async Task RemoveElementMetadataAsync(RedisKey key,
RedisValue field)
        {
            var metaKey = key.ToString() + Constants.TtlHashSuffix;
            await _database.HashDeleteAsync(metaKey, new RedisValue[]
            {
                $"{field}:frequency",
                $"{field}:lastAccess",
                $"{field}:lastFrequencyUpdateTime"
            });
            var sortedSetKey = key.ToString() +
Constants.TtlSortedSetSuffix;
            await _database.SortedSetRemoveAsync(sortedSetKey, field);

            string members = $"{key.ToString()}::{field}";
            await
_database.SortedSetRemoveAsync(Constants.GlobalFrequencySetKey, members);
            await
_database.SortedSetRemoveAsync(Constants.GlobalLastAccessSetKey, members);
            await
_database.SortedSetRemoveAsync(Constants.GlobalEvictionSetKey, members);
        }
    }
```

ДОДАТОК В

Код адаптивного менеджеру для заміщення кешу

```
using RedisCacheOptimizer.Configurations;
using RedisCacheOptimizer.Interfaces;
using RedisCacheOptimizer.Models;
using StackExchange.Redis;

namespace RedisCacheOptimizer;

public class AdaptiveCacheManager : BaseDatabaseClient,
IAdaptiveCacheManager
{
    private readonly AdaptiveCacheManagerOptions _options;
    private CancellationTokenSource _cancellationTokenSource = new();
    private const double EPSILON = 1e-9;

    private readonly object _evictionSetLock = new object();
    private readonly object _updateEvictionScoreLock = new object();
    private readonly SemaphoreSlim _cleanupSemaphore = new
SemaphoreSlim(1, 1);

    public int EvictionCount
    {
        get
        {
            return _evictionCount;
        }
    }

    private int _evictionCount;

    public AdaptiveCacheManager(string connectionString,
AdaptiveCacheManagerOptions options) : base(connectionString)
    {
        _options = options;

        _options.Validate();

        if (Math.Abs(_options.AgeWeight + _options.FrequencyWeight +
_options.MemoryWeight - 1.0) > 1e-9)
        {
            throw new ArgumentException("The sum of weight
coefficients must equal 1.");
        }
    }

    public async Task CleanupForRequiredMemoryAsync(long
requiredSize)
    {
        var server = _redis.GetServer(_redis.GetEndPoints()[0]);

        double memoryBufferPercentage = 0.15;
        long usedMemory = await GetUsedMemoryAsync(server);
        long maxMemory = await GetMaxMemoryAsync() ??
_options.MaxCacheMemory;
```

```
            long maxAllowedMemory = (long)(maxMemory * (1 -
memoryBufferPercentage));

            if (usedMemory + requiredSize <= maxAllowedMemory)
            {
                return;
            }

            long memoryToFree = (usedMemory + requiredSize) -
maxAllowedMemory;
            long freedSoFar = 0;

            while (memoryToFree > 0)
            {
                var batch = await
_database.SortedSetRangeByScoreWithScoresAsync(
                    Constants.GlobalEvictionSetKey,
                    start: double.NegativeInfinity,
                    stop: double.PositiveInfinity,
                    order: Order.Ascending,
                    take: 50
                );

                if (batch == null || batch.Length == 0)
                {
                    break;
                }

                foreach (var entry in batch)
                {
                    if (memoryToFree <= 0) break;

                    lock (_evictionSetLock)
                    {

_database.SortedSetRemove(Constants.GlobalEvictionSetKey, entry.Element);
                    }

                    string fullMember = entry.Element;
                    if (string.IsNullOrEmpty(fullMember)) continue;

                    string dataKey;
                    string propertyName = null;

                    if (fullMember.Contains("::"))
                    {
                        var parts = fullMember.Split(new[] { "::" },
StringSplitOptions.None);
                        dataKey = parts[0];
                        propertyName = parts[1];
                    }
                    else
                    {
                        dataKey = fullMember;
                    }
```

```
                    var isProtected = await
_database.StringGetAsync(dataKey + Constants.ProtectedSuffix);
                    if (isProtected == "true")
                    {
                        continue;
                    }

                    var metaKey = (RedisKey)(dataKey +
Constants.TtlHashSuffix);
                    var memoryUsageResult = await
_database.ExecuteAsync("MEMORY", "USAGE", dataKey);
                    if (!long.TryParse(memoryUsageResult.ToString(), out
var memorySize) || memorySize <= 0)
                    {
                        memorySize = 1;
                    }

                    if (!string.IsNullOrEmpty(propertyName))
                    {
                        await DeletePropertyAsync(dataKey, metaKey,
propertyName);
                    }
                    else
                    {
                        await DeleteKeyAsync(dataKey, metaKey);
                    }

                    freedSoFar += memorySize;
                    memoryToFree -= memorySize;
                    if (memoryToFree < 0)
                    {
                        memoryToFree = 0;
                    }
                }
            }

            if (memoryToFree > 0)
            {
                Console.WriteLine($"Warning: Unable to free enough
memory. Still need {memoryToFree} bytes.");
            }
            else
            {
                Console.WriteLine($"Freed {freedSoFar} bytes to
accommodate {requiredSize} bytes.");
            }
        }


        public void ClearEvictionCount()
        {
            _evictionCount = 0;
        }

        public void DisableCacheManager()
        {
            _cancellationTokenSource.Cancel();
        }
```

```
        public void EnableCacheManager()
        {
            _cancellationTokenSource = new();

            Task.Run(async () =>
            {
                while
(!_cancellationTokenSource.Token.IsCancellationRequested)
                {
                    try
                    {
                        await CleanupExpiredKeysAsync();
                        await Task.Delay(_options.CleanupInterval,
_cancellationTokenSource.Token);
                    }
                    catch (TaskCanceledException)
                    {
                    }
                    catch (Exception ex)
                    {
                        Log($"Error during CleanupExpiredKeys:
{ex.Message}");
                    }
                }
            }, _cancellationTokenSource.Token);
        }

        public async Task UpdateEvictionScoreAsync(string dataKey)
        {
            var metaKey = (RedisKey)(dataKey + Constants.TtlHashSuffix);
            var tracker = await GetObjectAccessTrackerAsync(metaKey);

            double now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();

            double freqGeneral = 0.0;
            double ageGeneral = 0.0;
            if (tracker.Frequency > 0)
            {
                double t = now - tracker.LastFrequencyUpdateTime;
                freqGeneral = tracker.Frequency * Math.Exp(-
_options.DecayLambda * t);
                ageGeneral = now - tracker.LastAccess;
            }

            var memoryUsageResult = await
_database.ExecuteAsync("MEMORY", "USAGE", dataKey);
            if (!long.TryParse(memoryUsageResult.ToString(), out var
memorySize) || memorySize <= 0)
            {
                memorySize = 1;
            }

            double fMax = await GetFMaxAsync();
            double aMax = await GetAMaxAsync();
            long mMax = _options.MaxCacheMemory;
```

```
            double sKey = ComputeEvictionScore(ageGeneral, freqGeneral,
memorySize, aMax, fMax, mMax);

            lock (_updateEvictionScoreLock)
            {
                _database.SortedSetAdd(Constants.GlobalEvictionSetKey,
dataKey, sKey);
            }

            int propCount = tracker.PropertyAccessState.Count;
            if (propCount > 0)
            {
                double propertyMemoryShare = memorySize / (propCount +
1.0);

                foreach (var kvp in tracker.PropertyAccessState)
                {
                    string propertyName = kvp.Key;
                    var propTracker = kvp.Value;

                    double freqProp = 0.0;
                    double ageProp = 0.0;
                    if (propTracker.Frequency > 0)
                    {
                        double tProp = now -
propTracker.LastFrequencyUpdateTime;
                        freqProp = propTracker.Frequency * Math.Exp(-
_options.DecayLambda * tProp);
                        ageProp = now - propTracker.LastAccess;
                    }

                    double sProp = ComputeEvictionScore(ageProp,
freqProp,

(long)propertyMemoryShare,
                                                        aMax, fMax,
mMax);

                    string zsetMember = $"{dataKey}::{propertyName}";

                    lock (_updateEvictionScoreLock)
                    {

_database.SortedSetAdd(Constants.GlobalEvictionSetKey, zsetMember, sProp);
                    }
                }
            }
        }

        private async Task CleanupExpiredKeysAsync()
        {
            await _cleanupSemaphore.WaitAsync();
            try
            {
                var server = _redis.GetServer(_redis.GetEndPoints()[0]);

                long usedMemory = await GetUsedMemoryAsync(server);
```

```
                long maxMemory = await GetMaxMemoryAsync() ??
_options.MaxCacheMemory;

                long memoryThreshold = (long)(_options.PMemoryThreshold *
maxMemory);

                if (usedMemory <= memoryThreshold)
                {
                    return;
                }

                long memoryToFree = usedMemory - memoryThreshold;
                long freedSoFar = 0;

                while (memoryToFree > 0)
                {
                    var batch = await
_database.SortedSetRangeByScoreWithScoresAsync(
                        Constants.GlobalEvictionSetKey,
                        start: double.NegativeInfinity,
                        stop: double.PositiveInfinity,
                        order: Order.Ascending,
                        take: 50
                    );

                    if (batch == null || batch.Length == 0)
                    {
                        break;
                    }

                    foreach (var entry in batch)
                    {
                        if (memoryToFree <= 0) break;

                        lock (_evictionSetLock)
                        {

_database.SortedSetRemove(Constants.GlobalEvictionSetKey, entry.Element);
                        }

                        string fullMember = entry.Element;
                        if (string.IsNullOrEmpty(fullMember)) continue;

                        string dataKey;
                        string propertyName = null;

                        if (fullMember.Contains("::"))
                        {
                            var parts = fullMember.Split(new[] { "::" },
StringSplitOptions.None);
                            dataKey = parts[0];
                            propertyName = parts[1];
                        }
                        else
                        {
                            dataKey = fullMember;
                        }
```

```
                            var isProtected = await
_database.StringGetAsync(dataKey + Constants.ProtectedSuffix);
                            if (isProtected == "true")
                            {
                                continue;
                            }

                            var metaKey = (RedisKey)(dataKey +
Constants.TtlHashSuffix);

                            var memoryUsageResult = await
_database.ExecuteAsync("MEMORY", "USAGE", dataKey);
                            if (!long.TryParse(memoryUsageResult.ToString(),
out var memorySize) || memorySize <= 0)
                            {
                                memorySize = 1;
                            }

                            if (!string.IsNullOrEmpty(propertyName))
                            {
                                await DeletePropertyAsync(dataKey, metaKey,
propertyName);
                            }
                            else
                            {
                                await DeleteKeyAsync(dataKey, metaKey);
                            }

                            freedSoFar += memorySize;
                            memoryToFree -= memorySize;
                            if (memoryToFree < 0)
                            {
                                memoryToFree = 0;
                            }
                        }
                    }

                    if (memoryToFree > 0)
                    {
                        Console.WriteLine($"[CleanupExpiredKeys] Unable to
free enough memory. Still need ~{memoryToFree} bytes.");
                    }
                    else
                    {
                        Console.WriteLine($"[CleanupExpiredKeys] Freed
~{freedSoFar} bytes; usedMemory was {usedMemory}, threshold is
{memoryThreshold}.");
                    }
                }
                finally
                {
                    _cleanupSemaphore.Release();
                }
            }

            private double ComputeEvictionScore(double age, double frequency,
long memorySize, double aMax, double fMax, long mMax)
            {
```

```
            double aNormalized = 0.0;
            if (aMax > 0)
            {
                aNormalized = Math.Min(age / aMax, 1.0);
            }

            double fNormalized = 0.0;
            if (fMax > 0)
            {
                fNormalized = Math.Min(frequency / fMax, 1.0);
            }

            double memInverted = (mMax > 0 && memorySize > 0)
                ? (mMax / (double)memorySize)
                : 1.0;

            double sAge = _options.AgeWeight * (1.0 / (aNormalized +
EPSILON));
            double sFreq = _options.FrequencyWeight * fNormalized;
            double sMem = _options.MemoryWeight * memInverted;

            double S = sAge + sFreq + sMem;
            return Math.Max(S, 0.0);
        }

        private async Task<long> GetUsedMemoryAsync(IServer server)
        {
            try
            {
                var info = await server.InfoAsync("memory");
                var memoryInfo = info.FirstOrDefault(x => x.Key ==
"Memory");
                var usedMemoryString = memoryInfo?.FirstOrDefault(entry
=> entry.Key == "used_memory").Value;
                if (long.TryParse(usedMemoryString, out var usedMemory))
                {
                    return usedMemory;
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine("Error retrieving used memory: " +
ex.Message);
            }
            return 0;
        }

        private async Task<long?> GetMaxMemoryAsync()
        {
            var server = _redis.GetServer(_redis.GetEndPoints()[0]);

            try
            {
                var config = await server.ConfigGetAsync("maxmemory");
                if (config != null && config.Length > 0)
                {
                    var maxMemoryString = config.FirstOrDefault().Value;
```

```
                    if (long.TryParse(maxMemoryString, out var
maxMemory))
                    {
                        return maxMemory > 0 ? maxMemory : long.MaxValue;
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine("Error retrieving max memory: " +
ex.Message);
            }
            return null;
        }

        private async Task<double> GetFMaxAsync()
        {
            var top = await
_database.SortedSetRangeByRankWithScoresAsync(
                Constants.GlobalFrequencySetKey,
                0,
                0,
                Order.Descending);

            if (top != null && top.Length > 0)
            {
                return top[0].Score;
            }
            return 0.0;
        }

        private async Task<double> GetAMaxAsync()
        {
            var now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();
            var bottom = await
_database.SortedSetRangeByRankWithScoresAsync(
                Constants.GlobalLastAccessSetKey,
                0,
                0,
                Order.Ascending);

            if (bottom != null && bottom.Length > 0)
            {
                double minLastAccess = bottom[0].Score;
                double aMax = now - minLastAccess;
                return (aMax > 0) ? aMax : 0.0;
            }
            return 0.0;
        }

        private async Task<ObjectAccessTracker>
GetObjectAccessTrackerAsync(RedisKey metaKey)
        {
            var objectAccessTracker = new ObjectAccessTracker();
            var entries = await _database.HashGetAllAsync(metaKey);

            foreach (var entry in entries)
            {
```

```
                string field = entry.Name;
                string value = entry.Value;

                if (field == "frequency")
                {
                    objectAccessTracker.Frequency = double.Parse(value);
                }
                else if (field == "lastAccess")
                {
                    objectAccessTracker.LastAccess = long.Parse(value);
                }
                else if (field == "lastFrequencyUpdateTime")
                {
                    objectAccessTracker.LastFrequencyUpdateTime =
long.Parse(value);
                }
                else if (field.Contains(":"))
                {
                    var parts = field.Split(':');
                    if (parts.Length == 2)
                    {
                        string propertyName = parts[0];
                        string propertyField = parts[1];

                        if
(!objectAccessTracker.PropertyAccessState.TryGetValue(propertyName, out var
tracker))
                        {
                            tracker = new AccessTracker();

objectAccessTracker.PropertyAccessState[propertyName] = tracker;
                        }

                        if (propertyField == "frequency")
                        {
                            tracker.Frequency = double.Parse(value);
                        }
                        else if (propertyField == "lastAccess")
                        {
                            tracker.LastAccess = long.Parse(value);
                        }
                        else if (propertyField ==
"lastFrequencyUpdateTime")
                        {
                            tracker.LastFrequencyUpdateTime =
long.Parse(value);
                        }
                    }
                }
            }

            return objectAccessTracker;
        }

        private async Task DeleteKeyAsync(string dataKey, RedisKey
metaKey)
        {
```

```
            var typeResult = await _database.ExecuteAsync("TYPE",
dataKey);
            string keyType = typeResult.ToString();

            if (keyType == "hash")
            {
                var deleteDataTask = _database.KeyDeleteAsync(dataKey);
                var deleteMetaTask = _database.KeyDeleteAsync(metaKey);
                await Task.WhenAll(deleteDataTask, deleteMetaTask);
            }
            else
            {
                await _database.KeyDeleteAsync(dataKey);
                await _database.KeyDeleteAsync(metaKey);
            }

            Interlocked.Increment(ref _evictionCount);

            await
_database.SortedSetRemoveAsync(Constants.GlobalFrequencySetKey, dataKey);
            await
_database.SortedSetRemoveAsync(Constants.GlobalLastAccessSetKey, dataKey);
            await
_database.SortedSetRemoveAsync(Constants.GlobalEvictionSetKey, dataKey);
        }

        private async Task DeletePropertyAsync(string dataKey, RedisKey
metaKey, string propertyName)
        {
            var typeResult = await _database.ExecuteAsync("TYPE",
dataKey);
            string keyType = typeResult.ToString();

            if (keyType == "hash")
            {
                var deleteDataTask = _database.HashDeleteAsync(dataKey,
propertyName);
                var deleteMetaTask = _database.HashDeleteAsync(metaKey,
new RedisValue[]
                {
                    $"{propertyName}:frequency",
                    $"{propertyName}:lastAccess",
                    $"{propertyName}:lastFrequencyUpdateTime"
                });
                await Task.WhenAll(deleteDataTask, deleteMetaTask);
            }
            else
            {
                await _database.KeyDeleteAsync(dataKey);
                await _database.KeyDeleteAsync(metaKey);
            }

            Interlocked.Increment(ref _evictionCount);

            string propertyMember = $"{dataKey}::{propertyName}";
```

```
            await
_database.SortedSetRemoveAsync(Constants.GlobalFrequencySetKey,
propertyMember);
            await
_database.SortedSetRemoveAsync(Constants.GlobalLastAccessSetKey,
propertyMember);
            await
_database.SortedSetRemoveAsync(Constants.GlobalEvictionSetKey,
propertyMember);
        }

        public void Dispose()
        {
        }
    }
```

ДОДАТОК Г

Презентаційний матеріал до кваліфікаційної роботи

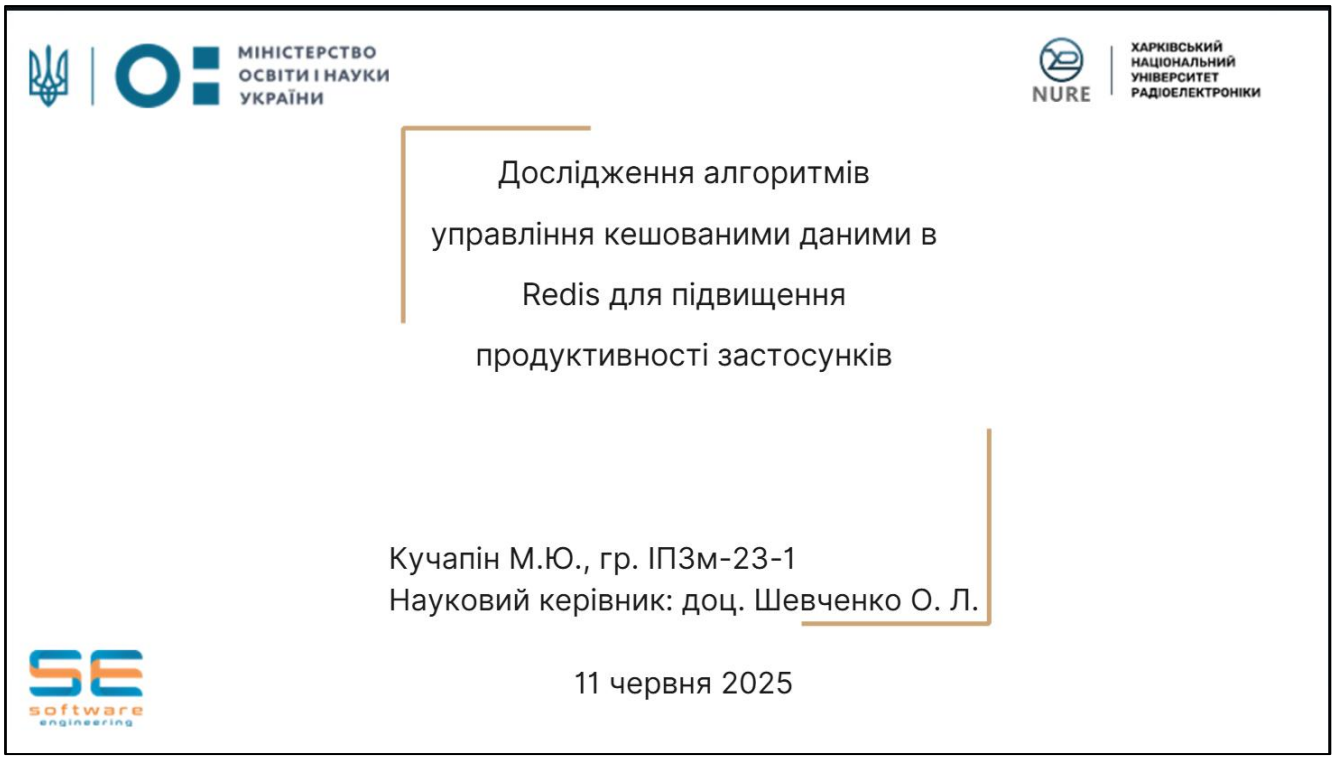


Рисунок Г.1 – Титульний аркуш (рисунок виконаний самостійно)

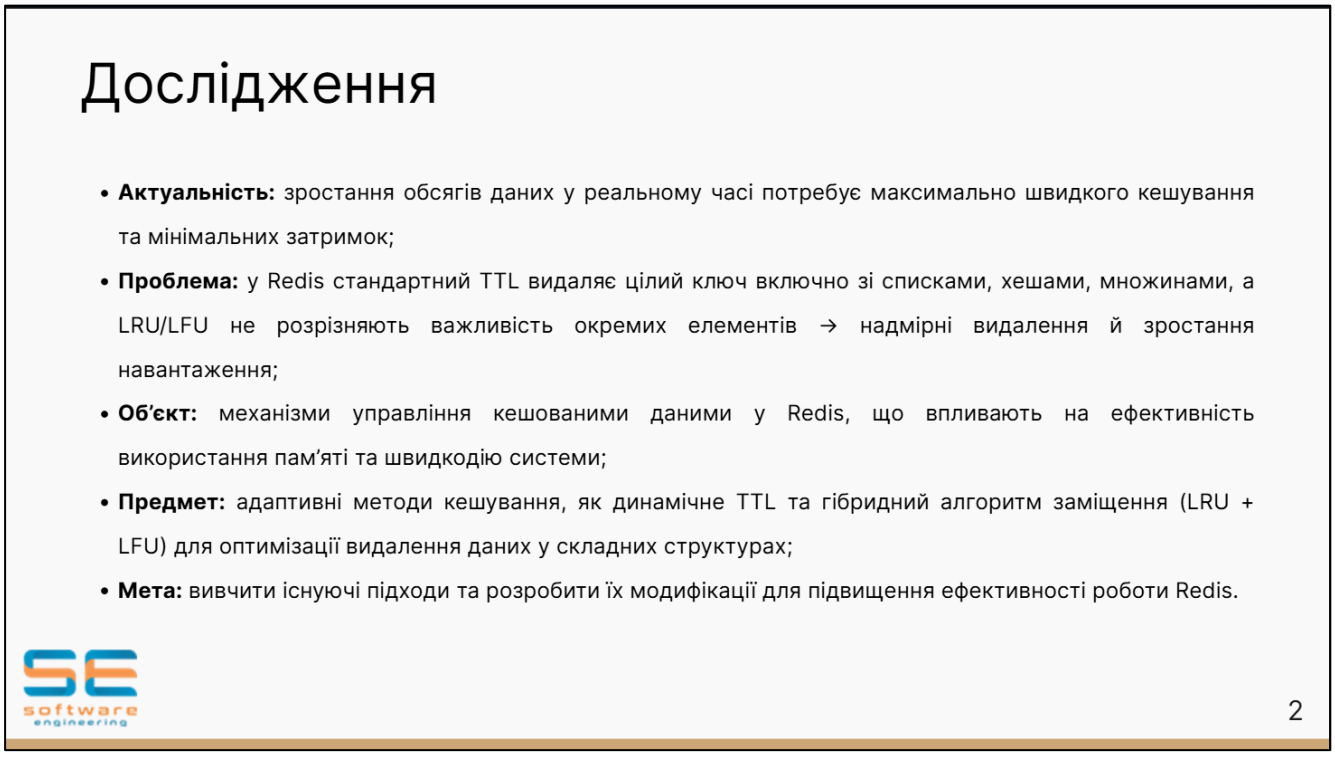


Рисунок Г.2 – Аркуш «Дослідження» (рисунок виконаний самостійно)

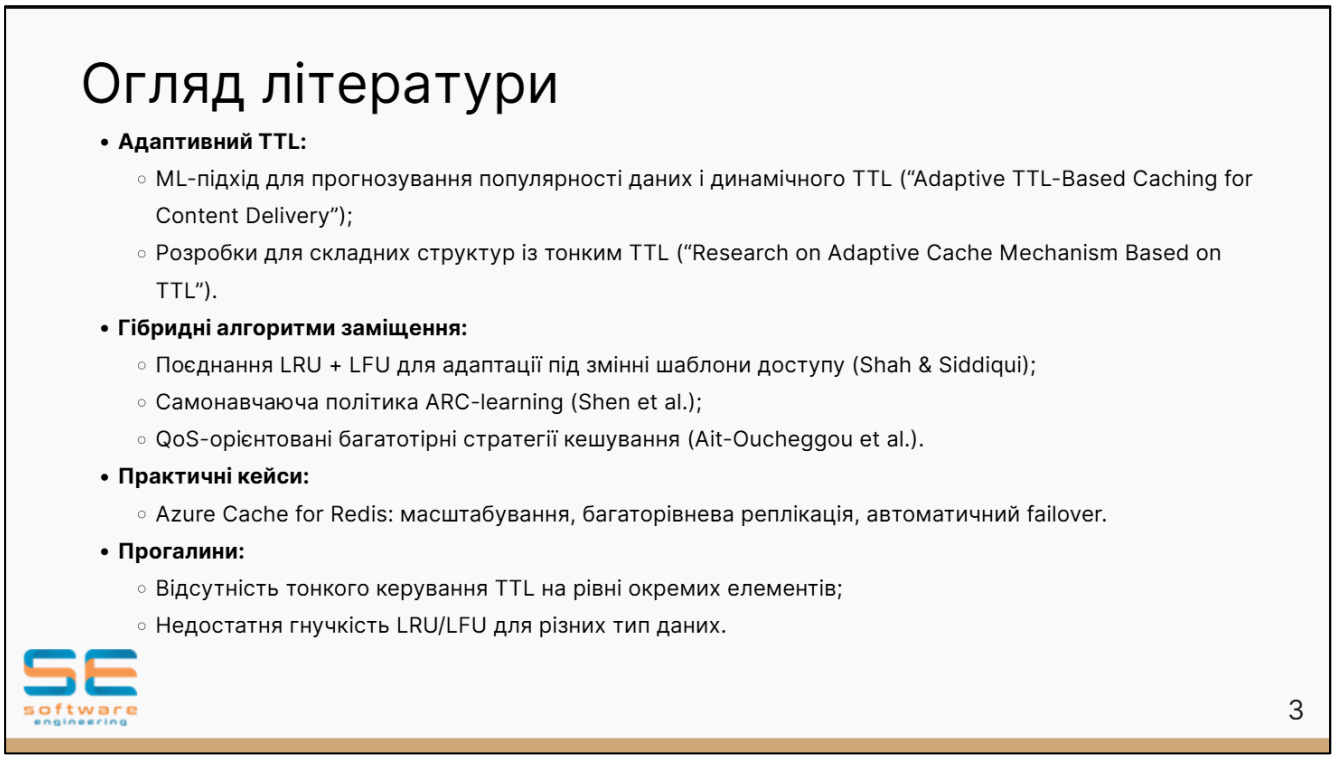


Рисунок Г.3 – Аркуш «Огляд літератури» (рисунок виконаний самостійно)

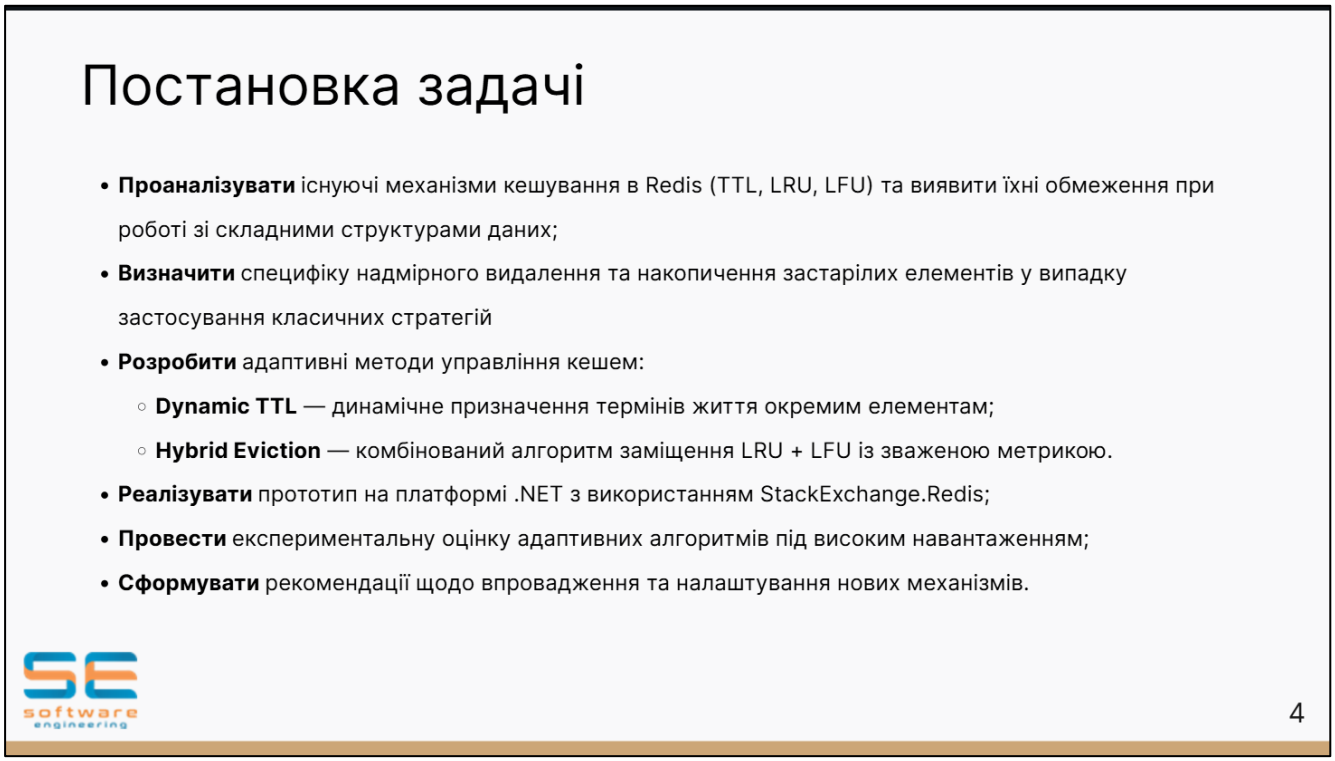


Рисунок Г.4 – Аркуш «Постановка задачі» (рисунок виконаний самостійно)

Методологія

- **Методи дослідження**
  - Теоретичний аналіз класичних механізмів кешування (TTL, LRU, LFU);
  - Математичне моделювання: формалізація **адаптивного TTL** та метрики **eviction-score (S)** з ваговими коефіцієнтами для давності, частоти доступу та розміру елемента;
  - Емпіричний метод: проектування, реалізація ПЗ та проведення експерименту.
- **Метрики оцінки й порівняння**
  - Рівень попадань і промахів кешу (Cache Hit Rate / Cache Miss Rate);
  - Кількість видалень по політиці (Evictions);
  - Середній TTL елементів (Average TTL);
  - Середній час запису/читання (Average Write and Read Times);
  - Δ використання пам'яті (Memory Usage Delta);
  - Використання CPU процесу (Process CPU Usage);
  - Δ пам'яті процесу (Process Memory Delta).

5

Рисунок Г.5 – Аркуш «Методологія» (рисунок виконаний самостійно)

Адаптивний TTL

- Експоненціальна модель розпаду підкреслює нещодавні звернення до даних:

$$f_d = f \times e^{-\lambda \times (t_c - t_l)}$$

- Нормована частота доступу забезпечує пропорційне масштабування:

$$f_n = min(\frac{f_d}{f_{max}}, 1)$$

- Адаптивне обчислення TTL балансує між збереженням даних і використанням пам'яті:

$$TTL = T_{base} + (T_{max} - T_{base}) \times f_n$$

6

Рисунок Г.6 – Аркуш «Адаптивний TTL» (рисунок виконаний самостійно)

Адаптивний алгоритм заміщення кешу

- Нормалізувати частоту доступу, як обговорювалося раніше;
- Нормалізувати вік доступу:

$$a = t_c - t_l \longrightarrow a_n = min(\frac{a}{a_{max}}, 1)$$

- Розрахунок балів за видалення (S):

$$S = w_a \times \frac{1}{a_n + \varepsilon} + w_f \times f_n + w_m \times \frac{m_{max}}{m}$$

- Вагові коефіцієнти повинні дорівнювати 1, щоб збалансувати вплив кожного показника на загальну оцінку S:

$$w_a + w_f + w_m = 1$$

software engineering

7

Рисунок Г.7 – Аркуш «Адаптивний алгоритм заміщення кешу» (рисунок виконаний самостійно)

Проведення експерименту

- **Прототип середовища:**
  - Реалізація на C# (.NET) з клієнтом StackExchange.Redis;
  - Контейнер Redis у Docker з обмеженням maxmemory та політикою витіснення.
- **Аппаратна платформа:**
  - Intel Core i7-12700H, 32 ГБ ОЗП, SSD;
- **Синтетичні дані:**
  - Об'єкти ~3 КБ у вигляді простих ключ-значення;
  - Набори від 10 000 до 100 000 записів.
- **Сценарії навантаження:**
  - "гарячі" та "холодні" ключі;
  - Консольний додаток для імітування роботи та збору метрик;
  - Повторювані ітерації для статистичної надійності.
- **Збір метрик;**
- **Аналіз результатів:**
  - Порівняння адаптивних та класичних стратегій за ключовими показниками.

software engineering

8

Рисунок Г.8 – Аркуш «Проведення експерименту» (рисунок виконаний самостійно)

Опис програмного забезпечення

- **Процес розробки**
  - Прототипування ключових компонентів, як Adaptive Cache Client, Adaptive Cache Manager та Policy Manager;
  - Моделювання архітектури;
  - Розробка ключових компонентів, як розширення з використанням розроблених підходів до існуючої бібліотеки;
  - Розробка додатку для симуляції навантаження та проведення експерименту;
  - Проведення модульного та інтеграційного тестування;
  - Проведення експерименту та збір метрик.
- **Вибрані мови програмування та фреймворки**
  - C# на платформі .NET;
  - StackExchange.Redis бібліотека для взаємодії з Redis;
  - Docker контейнер Redis для проведення експерименту;
  - XUnit для проведення модульного та інтеграційного тестування.

SE software engineering

9

Рисунок Г.9 – Аркуш «Опис програмного забезпечення» (рисунок виконаний самостійно)

UML-діаграма активності фонового процесу

SE software engineering

10

Рисунок Г.10 – Аркуш «UML-діаграма активності фонового процесу» (рисунок виконаний самостійно)

Рисунок Г.11 – Аркуш «UML-діаграма активності основного процесу» (рисунок виконаний самостійно)

Рисунок Г.12 – Аркуш «Архітектура системи» (рисунок виконаний самостійно)

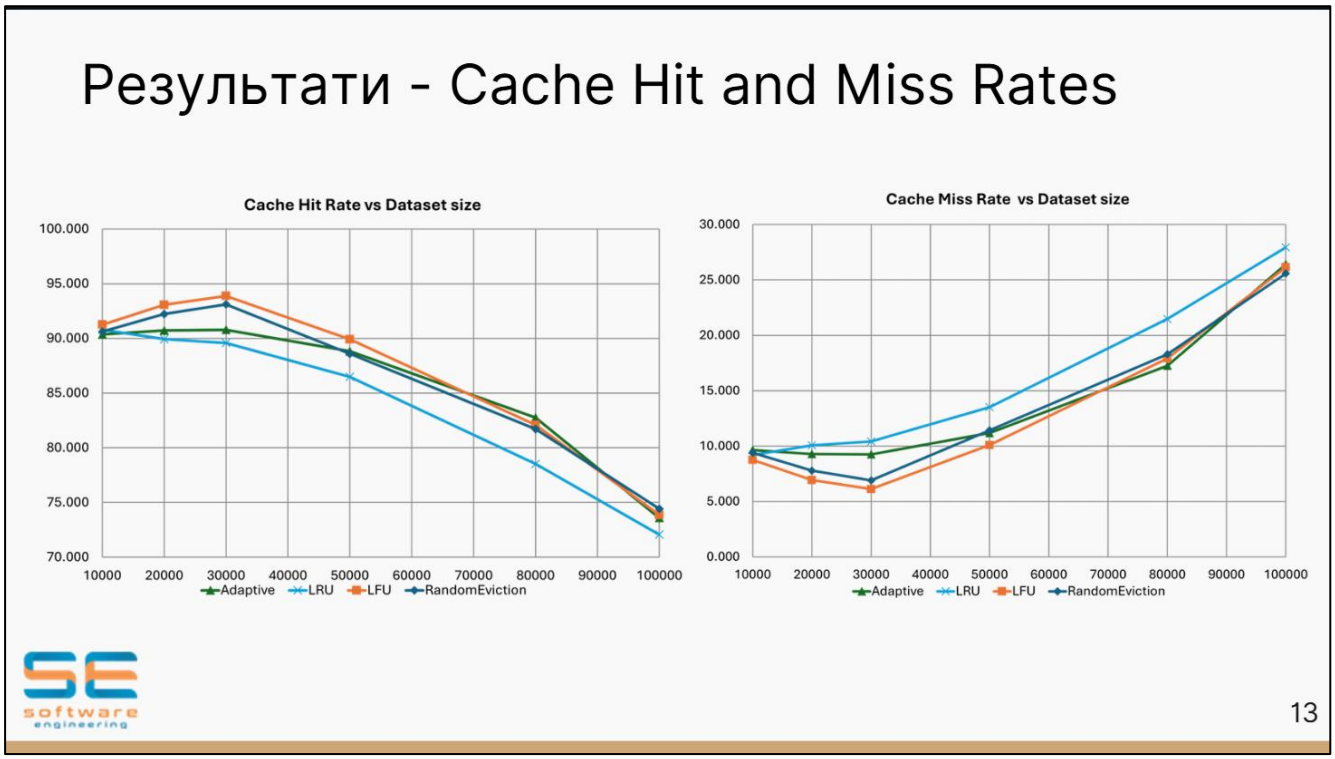


Рисунок Г.13 – Аркуш «Результати – Cache Hit and Miss Rates» (рисунок виконаний самостійно)

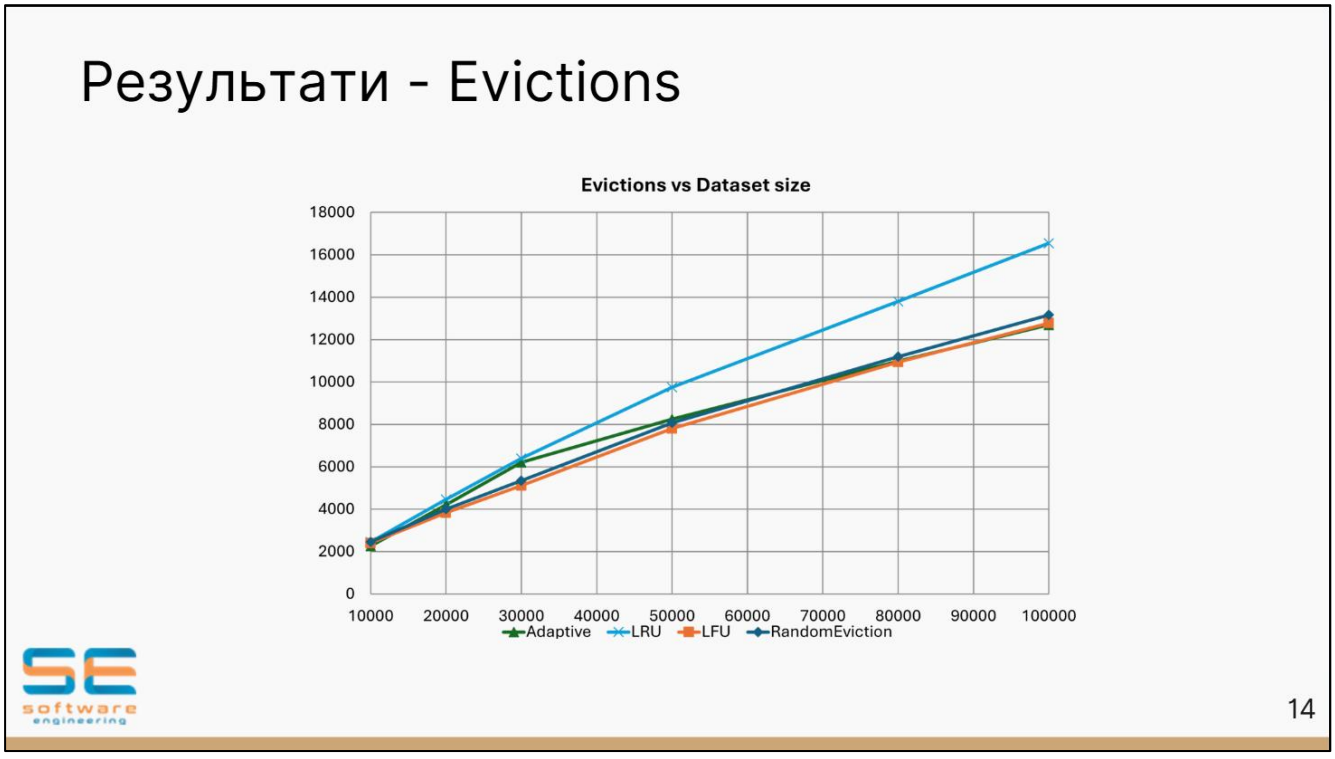


Рисунок Г.14 – Аркуш «Результати – Evictions» (рисунок виконаний самостійно)

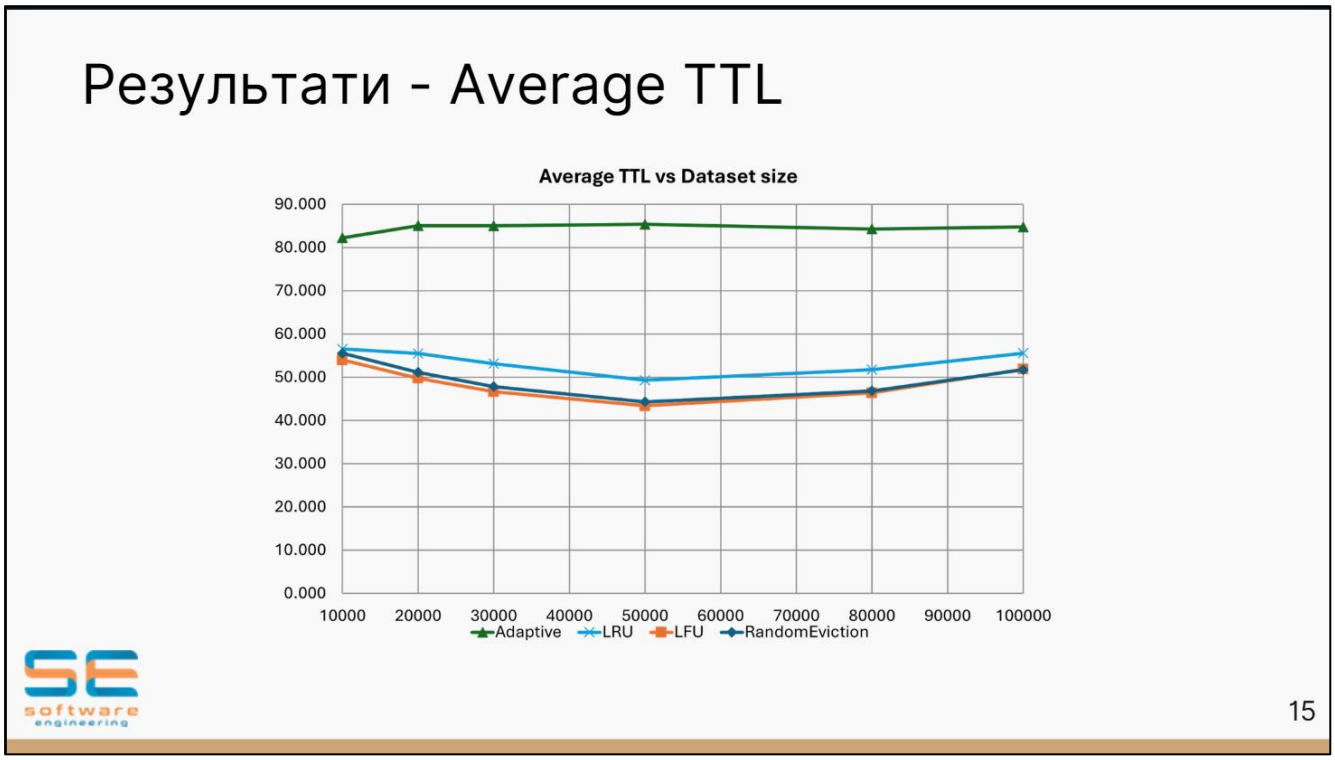


Рисунок Г.15 – Аркуш «Результати – Average TTL» (рисунок виконаний самостійно)

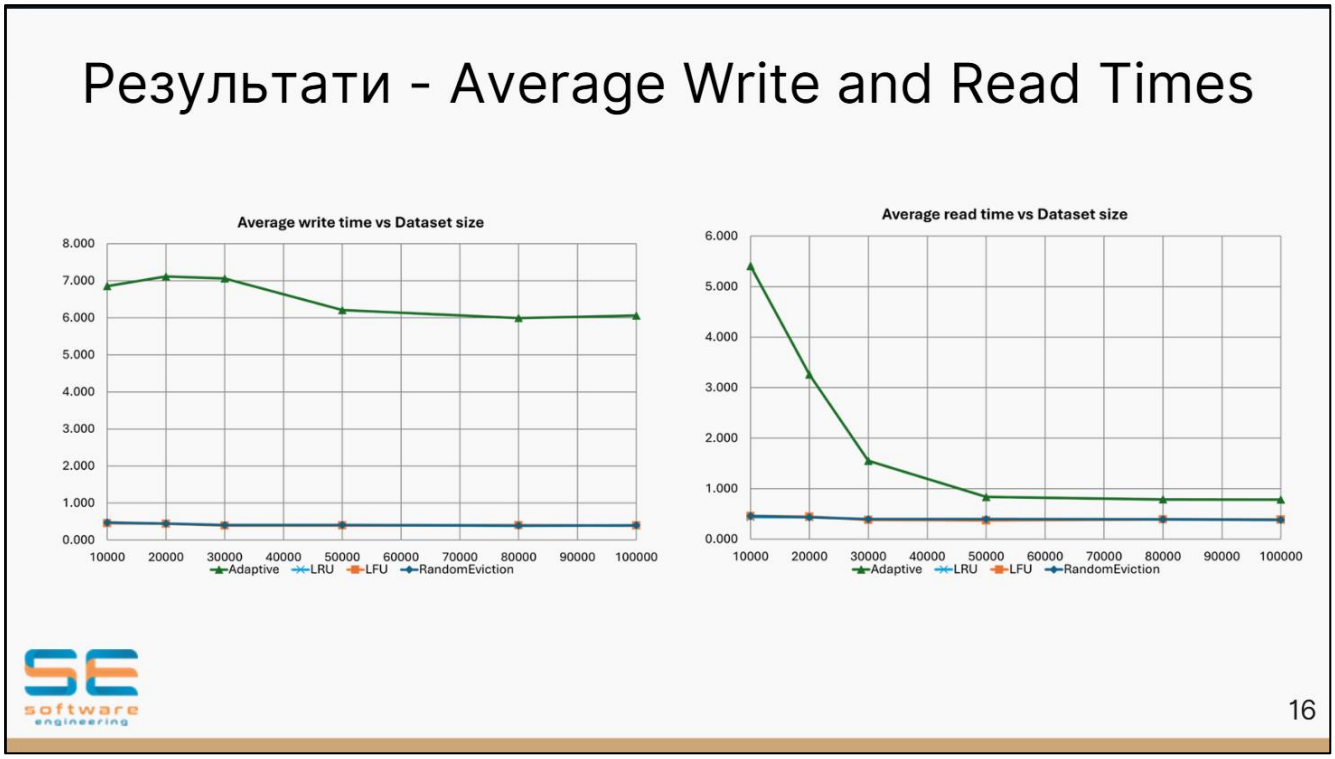


Рисунок Г.16 – Аркуш «Результати – Average Write and Read Times» (рисунок виконаний самостійно)

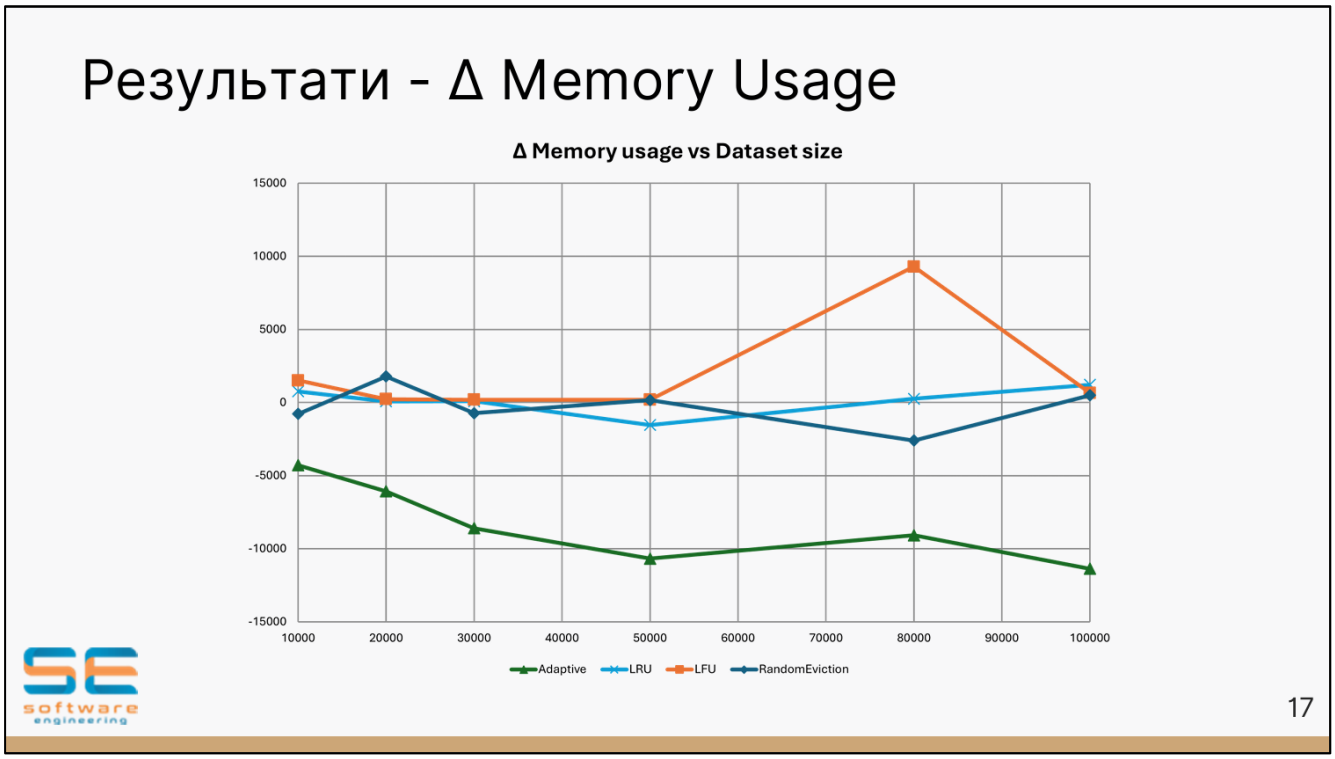


Рисунок Г.17 – Аркуш «Результати – Δ Memory Usage» (рисунок виконаний самостійно)

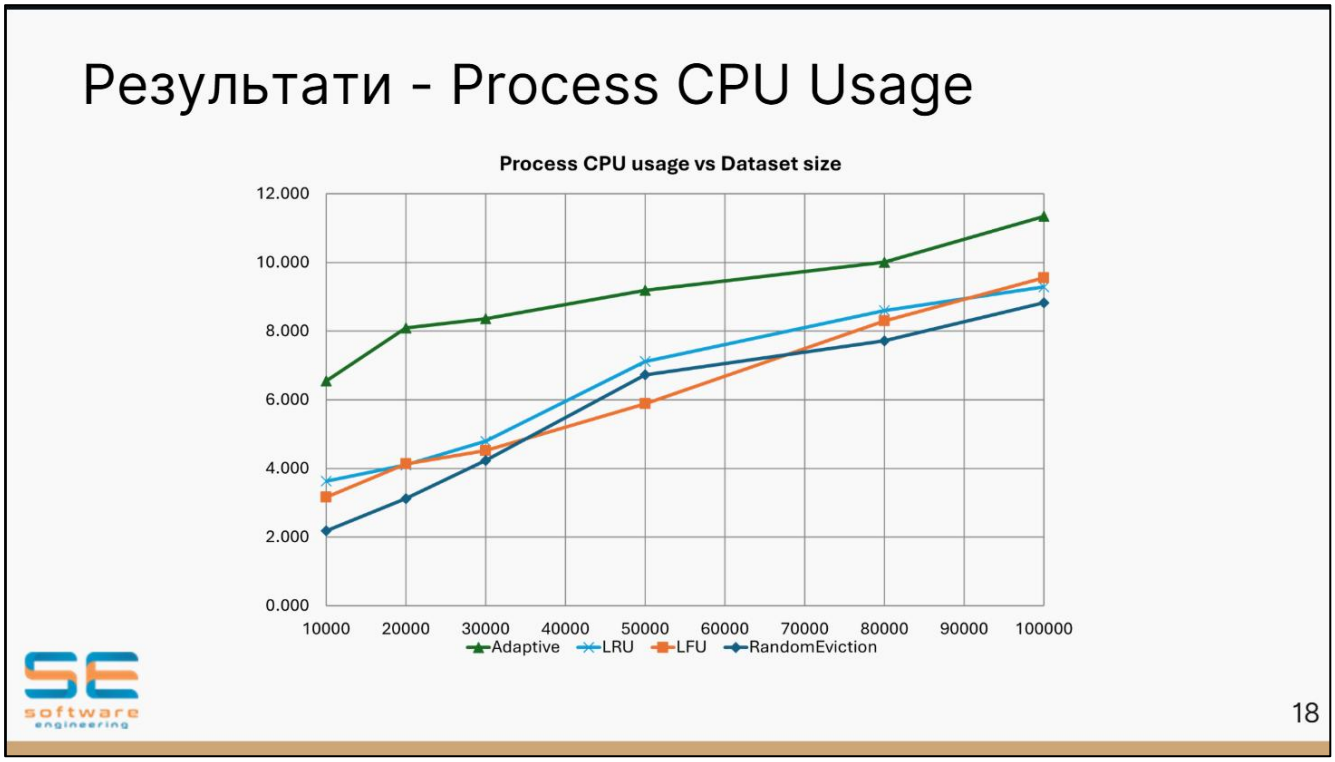


Рисунок Г.18 – Аркуш «Результати – Process CPU Usage» (рисунок виконаний самостійно)

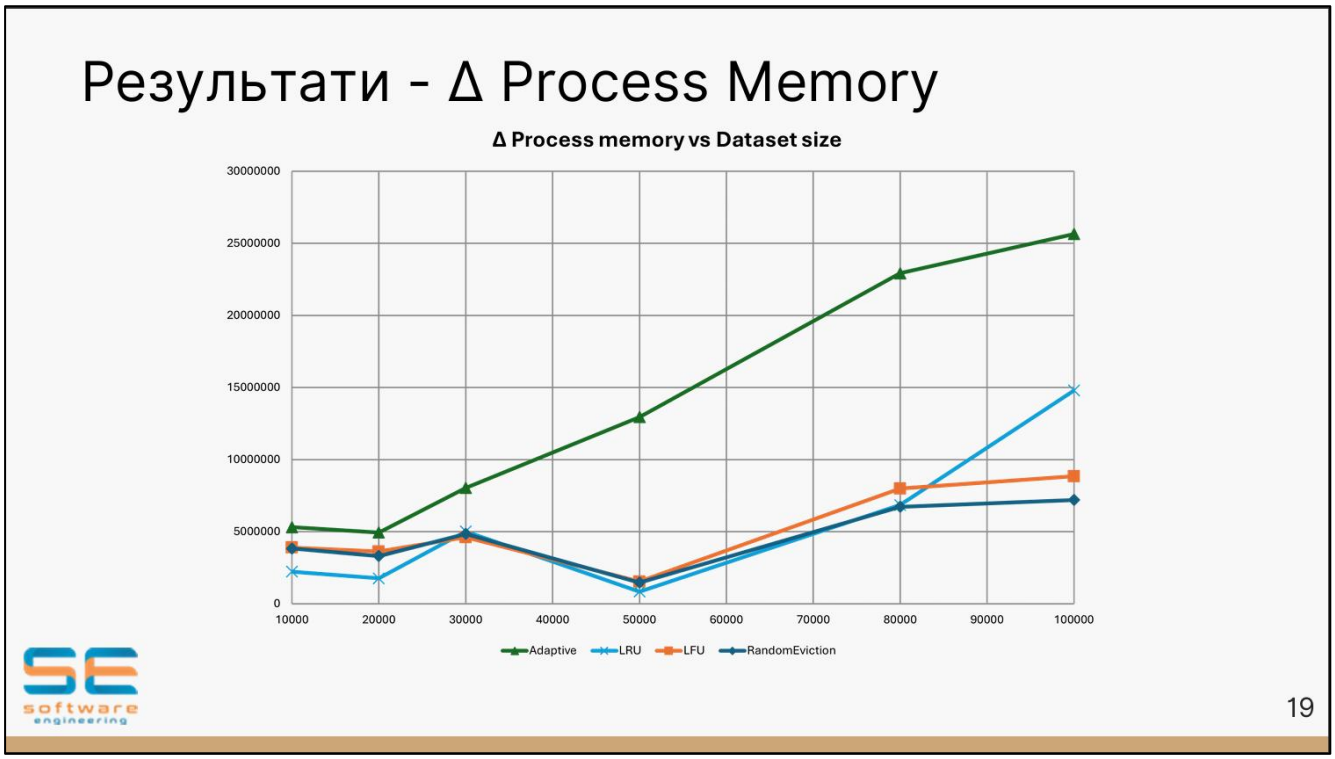


Рисунок Г.19 – Аркуш «Результати – Δ Process Memory» (рисунок виконаний самостійно)

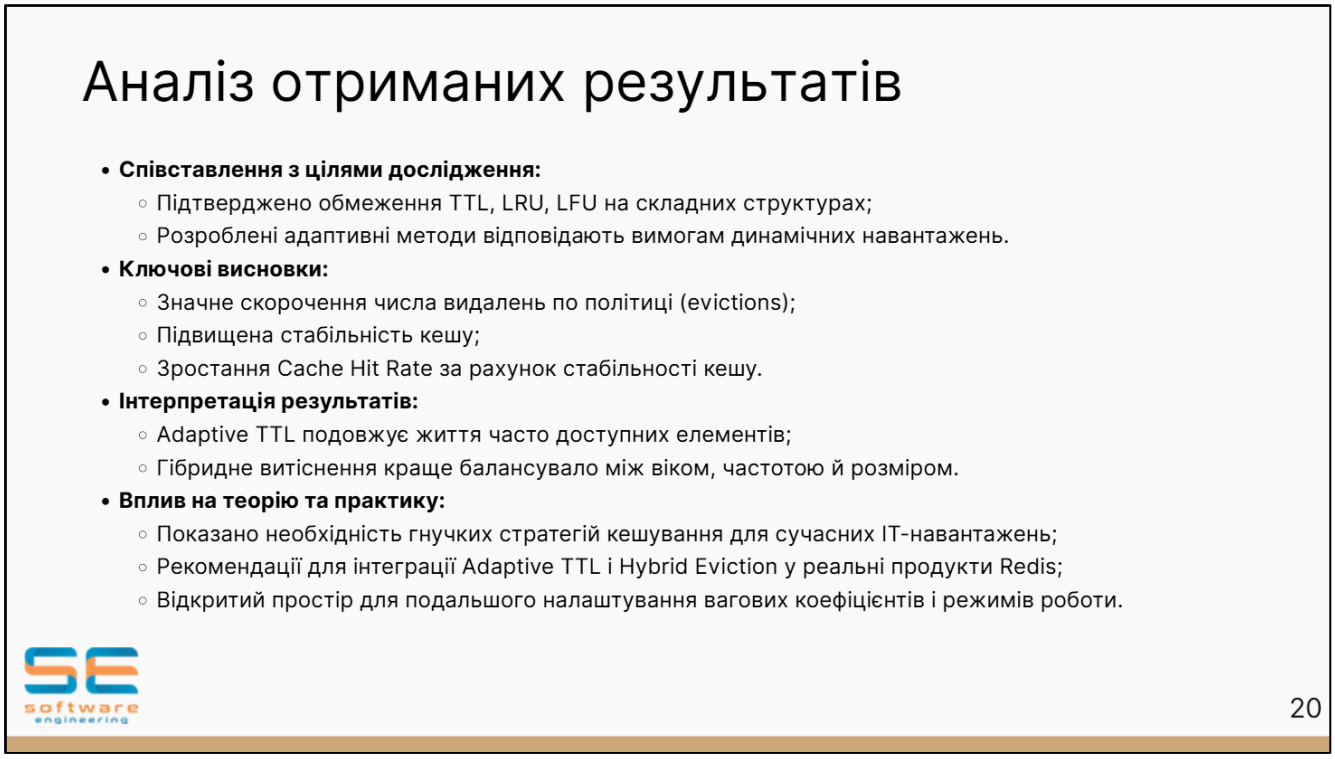


Рисунок Г.20 – Аркуш «Аналіз отриманих результатів» (рисунок виконаний самостійно)

Реалізовані покращення для роботи зі складними структурами

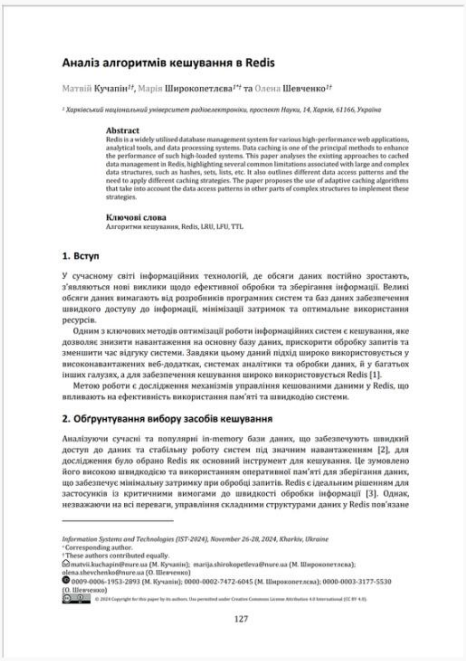
- Розроблено адаптивний підхід до встановлення TTL, що враховує особливості використання окремих елементів у складних структурах;
- Імплементовано гібридний алгоритм витіснення, який використовує інтегровану метрику пріоритету (S) для ефективнішого вибору кандидатів на видалення;
- Запроваджено механізм розбиття великих об'єктів на дрібніші компоненти з окремим TTL для кожного, що дозволяє уникнути втрати всіх елементів ключа одночасно та забезпечує гнучкіше керування кешем.

21

Рисунок Г.21 – Аркуш «Реалізовані покращення для роботи зі складними структурами» (рисунок виконаний самостійно)

Публікація результатів

Результати роботи були апробовані на науковій конференції – «13-та Міжнародна науково-технічна конференція «Інформаційні системи та технології ICT-2024» 26 - 28 листопада 2024 р», Харків, 2024



Аналіз алгоритмів кешування в Redis

Матвій Кучапін[1], Марія Широкопетлєва[1*] та Олена Шевченко[1*]

[1] *Харківський національний університет радіоелектроніки, проспект Науки, 14, Харків, 61166, Україна*

Abstract

Redis is a widely utilised database management system for various high-performance web applications, analytical tools, and data processing systems. Data caching is one of the principal methods to enhance the performance of such high-loaded systems. This paper analyses the existing approaches to cached data management in Redis, highlighting several common limitations associated with large and complex data structures, such as hashes, sets, lists, etc. It also outlines different data access patterns and the need to apply different caching strategies. The paper proposes the use of adaptive caching algorithms that take into account the data access patterns in other parts of complex structures to implement these strategies.

Ключові слова

Алгоритми кешування, Redis, LRU, LFU, TTL

1. Вступ

У сучасному світі інформаційних технологій, де обсяги даних постійно зростають, з'являються нові виклики щодо ефективної обробки та зберігання інформації. Великі обсяги даних вимагають від розробників програмних систем та баз даних забезпечення швидкого доступу до інформації, мінімізації затримок та оптимальне використання ресурсів.

Одним з ключових методів оптимізації роботи інформаційних систем є кешування, яке дозволяє знизити навантаження на основну базу даних, прискорити обробку запитів та зменшити час відгуку системи. Завдяки цьому даний підхід широко використовується у високонавантажених веб-додатках, системах аналітики та обробки даних, й у багатьох інших галузях, а для забезпечення кешування широко використовується Redis [1].

Метою роботи є дослідження механізмів управління кешованими даними у Redis, що впливають на ефективність використання пам'яті та швидкодію системи.

2. Обґрунтування вибору засобів кешування

Аналізуючи сучасні та популярні in-memory бази даних, що забезпечують швидкий доступ до даних та стабільну роботу систем під значним навантаженням [2], для дослідження було обрано Redis як основний інструмент для кешування. Це зумовлено його високою швидкодією та використанням оперативної пам'яті для зберігання даних, що забезпечує мінімальну затримку при обробці запитів. Redis є ідеальним рішенням для застосунків із критичними вимогами до швидкості обробки інформації [3]. Однак, незважаючи на всі переваги, управління складними структурами даних у Redis пов'язане

Information Systems and Technologies (IST-2024), November 26-28, 2024, Kharkiv, Ukraine

*Corresponding author.

†These authors contributed equally.

matvii.kuchapin@nure.ua (М. Кучапін); marija.shirokopetleva@nure.ua (М. Широкопетлєва); olena.shevchenko@nure.ua (О. Шевченко)

0009-0006-1953-2093 (М. Кучапін); 0000-0002-7472-6045 (М. Широкопетлєва); 0000-0003-3177-5530 (О. Шевченко)

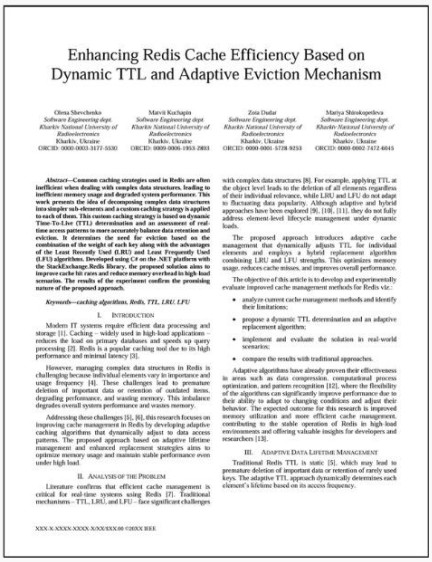
127

22

Рисунок Г.22 – Аркуш «Публікація результатів» (рисунок виконаний самостійно)

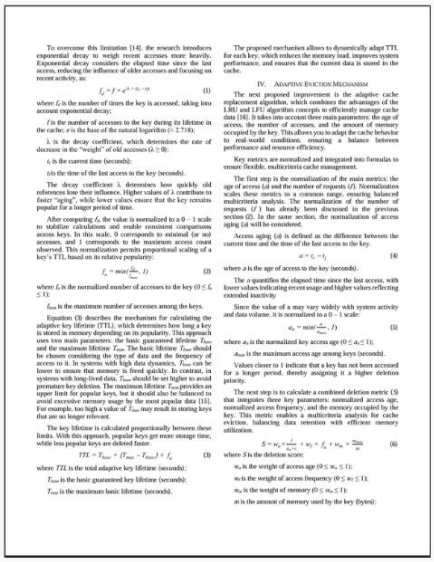
Публікація результатів

Результати роботи були апробовані на міжнародній конференції – «9th Open International Conference "Electrical, Electronic and Information Sciences" eStream 2025», Vilnius, Lithuania, 2025



Enhancing Redis Cache Efficiency Based on Dynamic TTL and Adaptive Eviction Mechanism

Olena Shevchenko, Software Engineering dept., Kharkiv National University of Radioelectronics, Kharkiv, Ukraine, ORCID: 0000-0003-3177-5530
Mátyás Kuchapin, Software Engineering dept., Kharkiv National University of Radioelectronics, Kharkiv, Ukraine, ORCID: 0009-0006-1863-2803
Zoia Dudar, Software Engineering dept., Kharkiv National University of Radioelectronics, Kharkiv, Ukraine, ORCID: 0000-0001-5728-9253
Mariya Shtonkopelieva, Software Engineering dept., Kharkiv National University of Radioelectronics, Kharkiv, Ukraine, ORCID: 0000-0002-7472-6043



23

Рисунок Г.23 – Аркуш «Публікація результатів» (рисунок виконаний самостійно)

Висновки

- **Адаптивне кешування перевершує традиційні методи:**
  - Покращена частота звернень до кешу та покращене використання пам’яті;
  - Підтримує стабільну продуктивність навіть в умовах високого навантаження.
- **Збалансований підхід:**
  - Адаптивний TTL довше зберігає дані, до яких часто звертаються.
  - Гібридне виселення (поєднання LRU і LFU) ефективно управляє актуальністю даних і витратами пам'яті.
- **Компроміси:**
  - Дещо більше навантаження на процесор і пам'ять під час запису;
  - Значне зменшення затримок при читанні та краще збереження кешу.
- **Подальші напрямки досліджень:**
  - Подальша оптимізація обчислювальних накладних витрат;
  - Дослідити адаптивні гібридні стратегії в реальному часі;
  - Точне налаштування вагових коефіцієнтів і параметрів TTL для конкретних профілів додатків.

24

Рисунок Г.24 – Аркуш «Висновки» (рисунок виконаний самостійно)

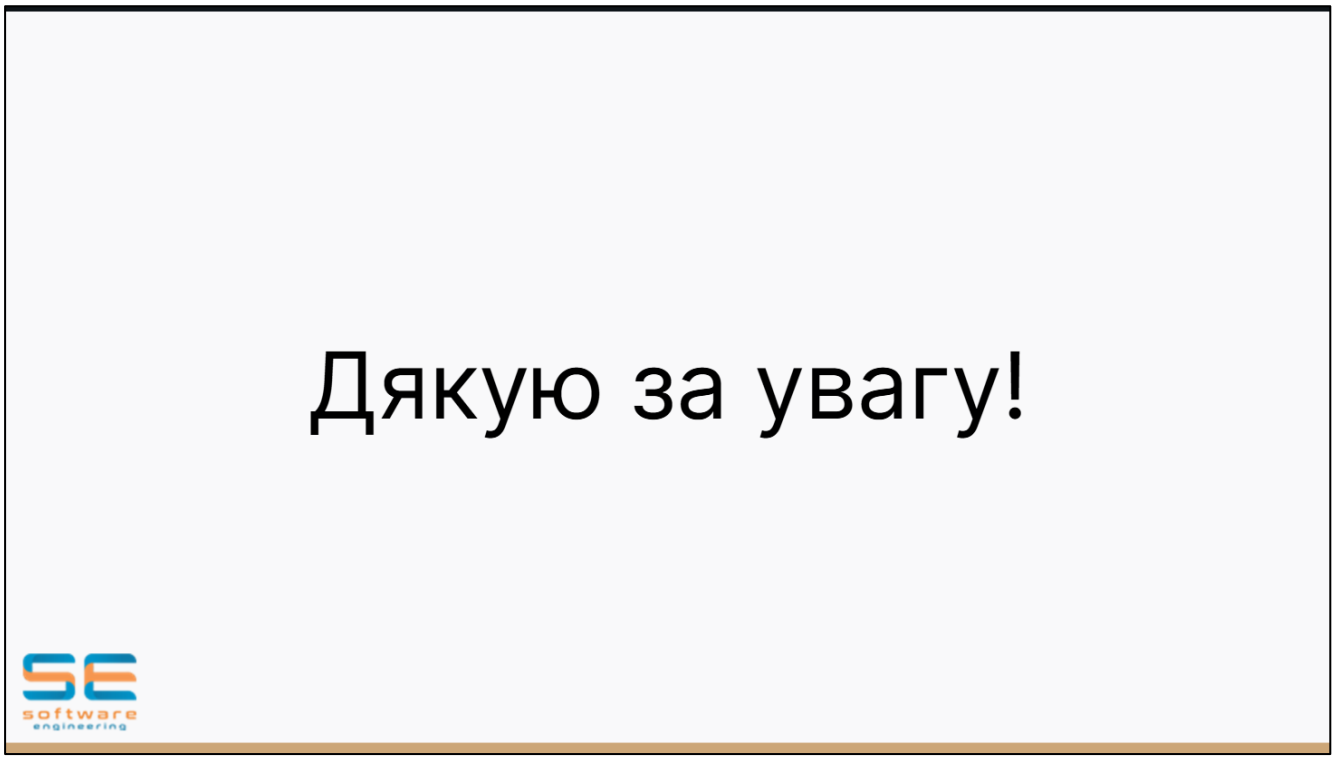
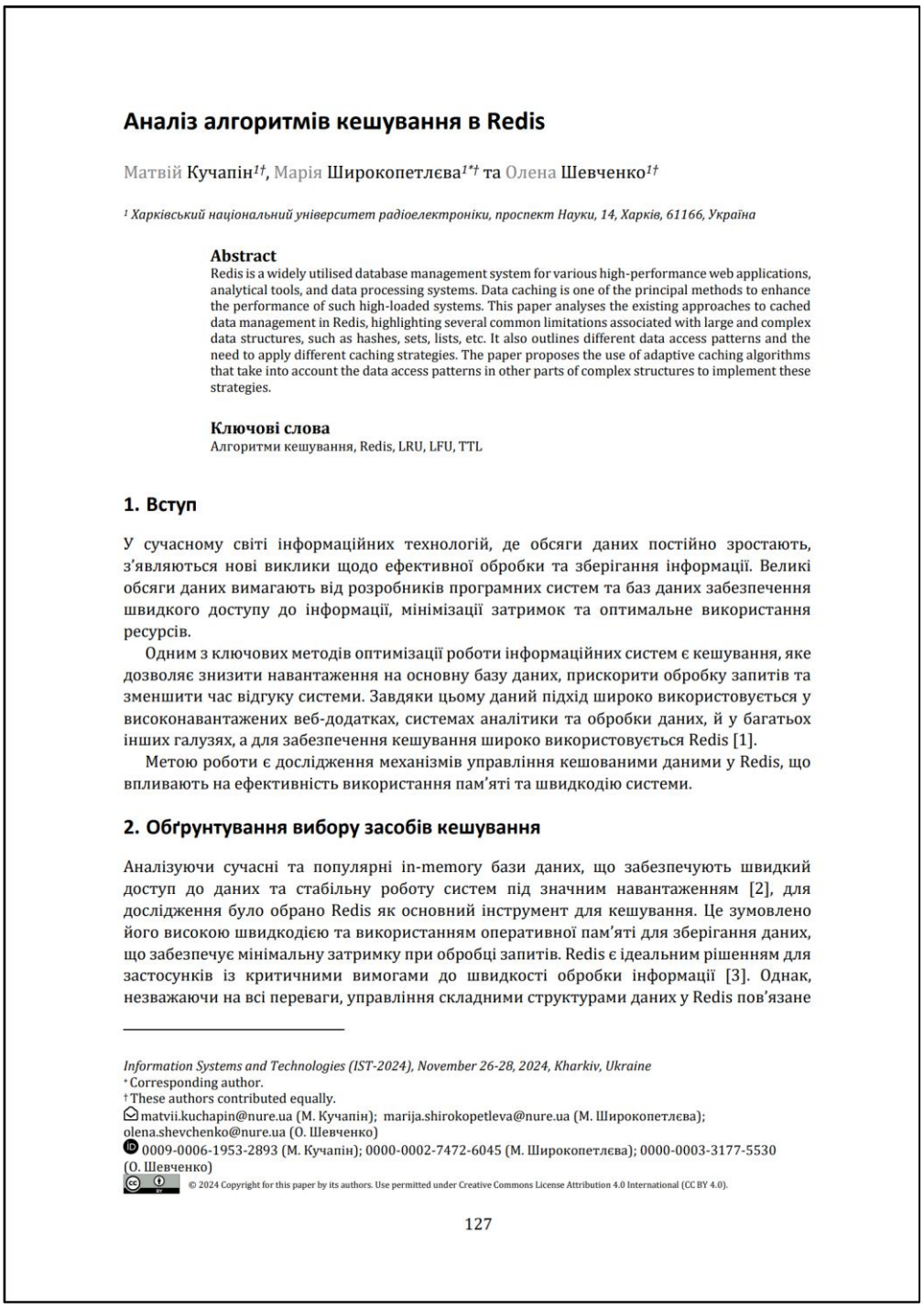


Рисунок Г.25 – Аркуш «Дякую за увагу!» (рисунок виконаний самостійно)

ДОДАТОК Д

Апробація результатів роботи на конференції «13-та Міжнародна науково-технічна конференція «Інформаційні системи та технології ІСТ-2024» 26 - 28 листопада 2024 р»



Аналіз алгоритмів кешування в Redis

Матвій Кучапін[1†], Марія Широкопетлєва[1*†] та Олена Шевченко[1†]

[1] Харківський національний університет радіоелектроніки, проспект Науки, 14, Харків, 61166, Україна

Abstract
Redis is a widely utilised database management system for various high-performance web applications, analytical tools, and data processing systems. Data caching is one of the principal methods to enhance the performance of such high-loaded systems. This paper analyses the existing approaches to cached data management in Redis, highlighting several common limitations associated with large and complex data structures, such as hashes, sets, lists, etc. It also outlines different data access patterns and the need to apply different caching strategies. The paper proposes the use of adaptive caching algorithms that take into account the data access patterns in other parts of complex structures to implement these strategies.

Ключові слова
Алгоритми кешування, Redis, LRU, LFU, TTL

1. Вступ

У сучасному світі інформаційних технологій, де обсяги даних постійно зростають, з'являються нові виклики щодо ефективної обробки та зберігання інформації. Великі обсяги даних вимагають від розробників програмних систем та баз даних забезпечення швидкого доступу до інформації, мінімізації затримок та оптимальне використання ресурсів.

Одним з ключових методів оптимізації роботи інформаційних систем є кешування, яке дозволяє знизити навантаження на основну базу даних, прискорити обробку запитів та зменшити час відгуку системи. Завдяки цьому даний підхід широко використовується у високонавантажених веб-додатках, системах аналітики та обробки даних, й у багатьох інших галузях, а для забезпечення кешування широко використовується Redis [1].

Метою роботи є дослідження механізмів управління кешованими даними у Redis, що впливають на ефективність використання пам'яті та швидкодію системи.

2. Обґрунтування вибору засобів кешування

Аналізуючи сучасні та популярні in-memory бази даних, що забезпечують швидкий доступ до даних та стабільну роботу систем під значним навантаженням [2], для дослідження було обрано Redis як основний інструмент для кешування. Це зумовлено його високою швидкодією та використанням оперативної пам'яті для зберігання даних, що забезпечує мінімальну затримку при обробці запитів. Redis є ідеальним рішенням для застосунків із критичними вимогами до швидкості обробки інформації [3]. Однак, незважаючи на всі переваги, управління складними структурами даних у Redis пов'язане

Information Systems and Technologies (IST-2024), November 26-28, 2024, Kharkiv, Ukraine
* Corresponding author.
† These authors contributed equally.
matvii.kuchapin@nure.ua (М. Кучапін); marija.shirokopetleva@nure.ua (М. Широкопетлєва); olena.shevchenko@nure.ua (О. Шевченко)
0009-0006-1953-2893 (М. Кучапін); 0000-0002-7472-6045 (М. Широкопетлєва); 0000-0003-3177-5530 (О. Шевченко)

Рисунок Д.1 – Перша сторінка статті (рисунок виконаний самостійно)

з низкою труднощів [4]. Проблеми виникають під час роботи з такими структурами, як хеші, множини і списки, де окремі елементи відрізняються за важливістю та частотою використання. У таких випадках надмірне видалення все ще корисних даних або, навпаки, зберігання старих елементів, які споживають ресурси пам'яті, може негативно позначитися на продуктивності системи. Це призводить до зниження загальної продуктивності через необхідність відтворювати або завантажувати дані повторно. Водночас зберігання застарілої інформації призводить до нераціонального використання пам'яті та знижує ефективність кешування.

Основною проблемою в контексті складних структур даних є відсутність гнучких механізмів управління на рівні окремих елементів усередині цих структур. Це обмежує можливість оптимізації роботи над великими об'єктами, які можуть містити як рідка використовувані, так і часто запитувані елементи.

3. Аналіз існуючих підходів до управління даними в Redis

У сучасних системах управління кешованими даними ключову роль відіграють алгоритми видалення та оновлення даних. Redis, як одна з найпопулярніших систем кешування, реалізує декілька підходів для управління даними: TTL, LRU та LFU. Однак, кожен із цих методів має свої обмеження при роботі зі складними структурами, що впливає на ефективність та продуктивність системи.

TTL є стандартним механізмом керування терміном життя даних у Redis [5]. Він дозволяє призначати певний термін існування кожному ключу, після закінчення якого дані автоматично видаляються з кешу. Це забезпечує контроль над свіжістю даних та автоматичне очищення кешу від застарілих записів. Проте, у випадку великих структур TTL застосовується до всього об'єкта. Це означає, що після завершення терміну життя видаляється вся структура, навіть якщо частина елементів усе ще актуальна. В результаті виникає проблема надмірного видалення корисних даних. Відсутність гнучкого управління TTL на рівні окремих елементів у великих структурах обмежує можливості адаптації системи до різних рівнів важливості та частоти використання даних.

Алгоритм LRU видаляє найдавніше використані елементи, щоб звільнити місце під нові дані [6]. Це дозволяє зберігати у кеші найчастіше використовувані елементи. Однак, при роботі зі складними структурами Redis не завжди коректно визначає, які елементи потрібно видаляти. У випадках, коли деякі елементи у структурі використовуються рідко, але залишаються важливими, LRU може видалити їх, що призводить до втрати критично важливих даних і зниження ефективності кешування. Відсутність розмежування важливості елементів у великих структурах робить цей алгоритм менш ефективним для складних випадків.

LFU орієнтується на частоту використання даних і видаляє найменш часто запитувані елементи [6]. Він більш ефективний для сценаріїв, де частота використання є важливішим критерієм, ніж час останнього доступу. Проте LFU також має свої обмеження при роботі зі складними структурами, тому що він не враховує зв'язки між елементами у структурах, що може призводити до видалення важливих даних, якщо вони запитуються нечасто, але є критичними для програми. Крім того, цей алгоритм потребує додаткових ресурсів для відстеження частоти використання, що ускладнює його реалізацію та знижує продуктивність при високих навантаженнях.

Існуючі підходи управління кешем у Redis мають кілька спільних обмежень, зокрема:

- неефективне використання пам'яті при застосуванні TTL до великих структур даних. Весь об'єкт видаляється після закінчення терміну життя, незалежно від активності окремих елементів, що призводить до перевитрати оперативної пам'яті та необхідності повторного створення кешованих даних;

Рисунок Д.2 – Друга сторінка статті (рисунок виконаний самостійно)

- відсутність можливості гнучкого управління TTL на рівні окремих елементів у великих структурах, що обмежує адаптивність системи до різних рівнів важливості та частоти використання даних. Це не дозволяє коректно видаляти застарілі елементи з великих структур, які можуть мати різну важливість;
- рідко використовувані елементи у великих структурах продовжують займати пам'ять навіть після того, як їх використання стає неактуальним.

В таблциці 1 наведено результати порівняння алгоритмів LRU та LFU.

Таблиця 5
Порівняльна таблиця алгоритмів LRU та LFU

Параметр	LRU	LFU
Критерій видалення	Час останнього доступу до ключа	Частота доступу до ключа
Переваги	Простота реалізації	Зберігає найпопулярніші ключі
Недоліки	Може видаляти часто використовувані, але нещодавно неактивні ключі	Повільна адаптація до змін у популярності
Підходить для	Динамічних патернів доступу	Стабільних патернів доступу

Дані характеристики вказують на те, що вибір між LFU та LRU залежить від конкретної системи та вимог до кешування. LRU є більш ефективним у системах з динамічними патернами доступу, де актуальність даних змінюється швидко. LFU, навпаки, підходить для систем зі стабільною популярністю даних, де певні ключі постійно користуються попитом. Різні патерни доступу до даних можуть впливати на ефективність кожного з цих алгоритмів, тому вибір алгоритму управління кешем повинен базуватися на аналізі поведінки користувачів та характеру даних у системі.

Загалом, використання стандартних механізмів управління кешем при роботі зі складними структурами даних може призвести до наступних негативних наслідків:

- старі та неактуальні дані можуть залишатися в пам'яті та займати місце, яке могло б бути використане для зберігання актуальної інформації;
- часте видалення та повторне створення великих структур даних збільшує навантаження на систему та призводить до збільшення часу відповіді;
- видалення актуальних даних через недосконаліть політик витіснення може призвести до неконсистентності даних та негативно вплинути на користувацький досвід.

При зростанні обсягів даних та кількості користувачів обмеження пам'яті та недостатня гнучкість політик витіснення стають більш помітними, що може обмежити можливості систем.

Одним із перспективних напрямків є розробка динамічних механізмів TTL, які дозволять контролювати час життя даних на рівні окремих елементів складних структур, враховуючи їхню активність та важливість.

Крім того, сучасні дослідження орієнтуються на створення алгоритмів, що поєднують можливості LRU та LFU, але мають покращену гнучкість і здатність адаптуватися до змінних умов роботи системи [7]. Такий підхід дозволяє краще управляти пам'яттю, зберігаючи часто запитувані дані та своєчасно видаляючи рідко використовувані елементи, знижуючи загальну затримку доступу до кешованих даних.

Рисунок Д.3 – Третя сторінка статті (рисунок виконаний самостійно)

Перспективи розвитку полягають у впровадженні комбінованих алгоритмів, які автоматично підлаштовуються під поточне навантаження та обсяг кешованої інформації [8]. Подальша інтеграція таких адаптивних алгоритмів у кешуючі системи, зокрема Redis, дозволить ефективніше вирішувати проблеми масштабування кешованих даних, підвищувати продуктивність та забезпечувати високу швидкість доступу до даних у системах із високим навантаженням.

Одним із важливих напрямків у сучасних дослідженнях є адаптивні алгоритми кешування, які здатні пристосовуватися до динамічних патернів доступу до даних.

4. Висновки

Недосконалість стандартних підходів також проявляється у неможливості тонкого налаштування під конкретні потреби додатків. Стандартні політики витіснення не враховують особливості доступу до даних у різних частинах складних структур. Це може обмежити ефективність кешування та призвести до нераціонального використання ресурсів.

Таким чином, стандартні підходи TTL, LRU та LFU мають суттєві недоліки при роботі зі складними структурами даних у високопродуктивних системах. Для подолання цих проблем необхідно розробити більш гнучкі та адаптивні механізми управління кешем, які враховують специфіку складних структур даних та характер доступу до них, забезпечуючи ефективне використання ресурсів та високу продуктивність системи, що є напрямком подальших досліджень.

Перелік посилань

[1] Redis Docs. 2024. URL: https://redis.io/docs/latest/ (дата звернення 12.10.2024).
[2] Top 27 In-Memory Databases Compared. 2024. URL: https://www.dragonflydb.io/guides/in-memory-databases.
[3] A. Kuzochkina, M. Shirokopetleva and Z. Dudar, Analyzing and Comparison of NoSQL DBMS, 2018 International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T), Kharkiv, Ukraine, 2018, pp. 560-564, doi: 10.1109/INFOCOMMST.2018.8632133.
[4] P. Zhang, et al, Redis++: A High Performance In-Memory Database Based on Segmented Memory Management and Two-Level Hash Index, 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), Melbourne, VIC, Australia, 2018, pp. 840-847, doi: 10.1109/BDCloud.2018.00125.
[5] J. Liu, X. Wan, Q. Zhu, T. Peng and X. Hu, Research on Adaptive Cache Mechanism Based on TTL, 2022 2nd International Conference on Networking, Communications and Information Technology (NetCIT), Manchester, United Kingdom, 2022, pp. 507-511, doi: 10.1109/NetCIT57419.2022.00125.
[6] Z. Li, D. Liu and H. Bi, CRFP: A Novel Adaptive Replacement Policy Combined the LRU and LFU Policies, 2008 IEEE 8th International Conference on Computer and Information Technology Workshops, Sydney, NSW, Australia, 2008, pp. 72-79, doi: 10.1109/CIT.2008.Workshops.22.
[7] W. Ma, Y. Zhu, S. Wang and Y. Bao, LearnedCache: A Locality-Aware Collaborative Data Caching by Learning Model, 2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), Xiamen, China, 2019, pp. 718-726, doi: 10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00109.

Рисунок Д.4 – Четверта сторінка статті (рисунок виконаний самостійно)

ДОДАТОК Е

Апробація результатів роботи на конференції «9th Open International Conference "Electrical, Electronic and Information Sciences“ eStream 2025»

Enhancing Redis Cache Efficiency Based on Dynamic TTL and Adaptive Eviction Mechanism

Olena Shevchenko
Software Engineering dept.
Kharkiv National University of Radioelectronics
Kharkiv, Ukraine
ORCID: 0000-0003-3177-5530

Matvii Kuchapin
Software Engineering dept.
Kharkiv National University of Radioelectronics
Kharkiv, Ukraine
ORCID: 0009-0006-1953-2893

Zoia Dudar
Software Engineering dept.
Kharkiv National University of Radioelectronics
Kharkiv, Ukraine
ORCID: 0000-0001-5728-9253

Mariya Shirokopetleva
Software Engineering dept.
Kharkiv National University of Radioelectronics
Kharkiv, Ukraine
ORCID: 0000-0002-7472-6045

***Abstract*—Common caching strategies used in Redis are often inefficient when dealing with complex data structures, leading to inefficient memory usage and degraded system performance. This work presents the idea of decomposing complex data structures into simpler sub-elements and a custom caching strategy is applied to each of them. This custom caching strategy is based on dynamic Time-To-Live (TTL) determination and an assessment of real-time access patterns to more accurately balance data retention and eviction. It determines the need for eviction based on the combination of the weight of each key along with the advantages of the Least Recently Used (LRU) and Least Frequently Used (LFU) algorithms. Developed using C# on the .NET platform with the StackExchange.Redis library, the proposed solution aims to improve cache hit rates and reduce memory overhead in high-load scenarios. The results of the experiment confirm the promising nature of the proposed approach.**

Keywords—caching algorithms, Redis, TTL, LRU, LFU

I. Introduction

Modern IT systems require efficient data processing and storage [1]. Caching – widely used in high-load applications – reduces the load on primary databases and speeds up query processing [2]. Redis is a popular caching tool due to its high performance and minimal latency [3].

However, managing complex data structures in Redis is challenging because individual elements vary in importance and usage frequency [4]. These challenges lead to premature deletion of important data or retention of outdated items, degrading performance, and wasting memory. This imbalance degrades overall system performance and wastes memory.

Addressing these challenges [5], [6], this research focuses on improving cache management in Redis by developing adaptive caching algorithms that dynamically adjust to data access patterns. The proposed approach based on adaptive lifetime management and enhanced replacement strategies aims to optimize memory usage and maintain stable performance even under high load.

II. Analysis of the Problem

Literature confirms that efficient cache management is critical for real-time systems using Redis [7]. Traditional mechanisms – TTL, LRU, and LFU – face significant challenges with complex data structures [8]. For example, applying TTL at the object level leads to the deletion of all elements regardless of their individual relevance, while LRU and LFU do not adapt to fluctuating data popularity. Although adaptive and hybrid approaches have been explored [9], [10], [11], they do not fully address element-level lifecycle management under dynamic loads.

The proposed approach introduces adaptive cache management that dynamically adjusts TTL for individual elements and employs a hybrid replacement algorithm combining LRU and LFU strengths. This optimizes memory usage, reduces cache misses, and improves overall performance.

The objective of this article is to develop and experimentally evaluate improved cache management methods for Redis viz.:

- analyze current cache management methods and identify their limitations;
- propose a dynamic TTL determination and an adaptive replacement algorithm;
- implement and evaluate the solution in real-world scenarios;
- compare the results with traditional approaches.

Adaptive algorithms have already proven their effectiveness in areas such as data compression, computational process optimization, and pattern recognition [12], where the flexibility of the algorithms can significantly improve performance due to their ability to adapt to changing conditions and adjust their behavior. The expected outcome for this research is improved memory utilization and more efficient cache management, contributing to the stable operation of Redis in high-load environments and offering valuable insights for developers and researchers [13].

III. Adaptive Data Lifetime Management

Traditional Redis TTL is static [5], which may lead to premature deletion of important data or retention of rarely used keys. The adaptive TTL approach dynamically determines each element’s lifetime based on its access frequency.

Рисунок Е.1 – Перша сторінка статті (рисунок виконаний самостійно)

To overcome this limitation [14], the research introduces exponential decay to weigh recent accesses more heavily. Exponential decay considers the elapsed time since the last access, reducing the influence of older accesses and focusing on recent activity, as:

$$f_d = f \times e^{-\lambda \times (t_c - t_l)} \quad (1)$$

where f_d is the number of times the key is accessed, taking into account exponential decay;

f is the number of accesses to the key during its lifetime in the cache; e is the base of the natural logarithm (≈ 2.718);

λ is the decay coefficient, which determines the rate of decrease in the "weight" of old accesses ($\lambda \geq 0$);

t_c is the current time (seconds);

t_l is the time of the last access to the key (seconds).

The decay coefficient λ determines how quickly old references lose their influence. Higher values of λ contribute to faster "aging", while lower values ensure that the key remains popular for a longer period of time.

After computing f_d, the value is normalized to a 0 – 1 scale to stabilize calculations and enable consistent comparisons across keys. In this scale, 0 corresponds to minimal (or no) accesses, and 1 corresponds to the maximum access count observed. This normalization permits proportional scaling of a key's TTL based on its relative popularity:

$$f_n = min(\frac{f_d}{f_{max}}, 1) \quad (2)$$

where f_n is the normalized number of accesses to the key ($0 \leq f_n \leq 1$);

f_{max} is the maximum number of accesses among the keys.

Equation (3) describes the mechanism for calculating the adaptive key lifetime (TTL), which determines how long a key is stored in memory depending on its popularity. This approach uses two main parameters: the basic guaranteed lifetime T_{base} and the maximum lifetime T_{max}. The basic lifetime T_{base} should be chosen considering the type of data and the frequency of access to it. In systems with high data dynamics, T_{base} can be lower to ensure that memory is freed quickly. In contrast, in systems with long-lived data, T_{base} should be set higher to avoid premature key deletion. The maximum lifetime T_{max} provides an upper limit for popular keys, but it should also be balanced to avoid excessive memory usage by the most popular data [15]. For example, too high a value of T_{max} may result in storing keys that are no longer relevant.

The key lifetime is calculated proportionally between these limits. With this approach, popular keys get more storage time, while less popular keys are deleted faster.

$$TTL = T_{base} + (T_{max} - T_{base}) \times f_n \quad (3)$$

where TTL is the total adaptive key lifetime (seconds);

T_{base} is the basic guaranteed key lifetime (seconds);

T_{max} is the maximum basic lifetime (seconds).

The proposed mechanism allows to dynamically adapt TTL for each key, which reduces the memory load, improves system performance, and ensures that the current data is stored in the cache.

IV. Adaptive Eviction Mechanism

The next proposed improvement is the adaptive cache replacement algorithm, which combines the advantages of the LRU and LFU algorithm concepts to efficiently manage cache data [16]. It takes into account three main parameters: the age of access, the number of accesses, and the amount of memory occupied by the key. This allows you to adapt the cache behavior to real-world conditions, ensuring a balance between performance and resource efficiency.

Key metrics are normalized and integrated into formulas to ensure flexible, multicriteria cache management.

The first step is the normalization of the main metrics: the age of access (a) and the number of requests (f). Normalization scales these metrics to a common range, ensuring balanced multicriteria analysis. The normalization of the number of requests (f) has already been discussed in the previous section (2). In the same section, the normalization of access aging (a) will be considered.

Access aging (a) is defined as the difference between the current time and the time of the last access to the key.

$$a = t_c - t_l \quad (4)$$

where a is the age of access to the key (seconds).

The a quantifies the elapsed time since the last access, with lower values indicating recent usage and higher values reflecting extended inactivity

Since the value of a may vary widely with system activity and data volume, it is normalized to a 0 – 1 scale:

$$a_n = min(\frac{a}{a_{max}}, 1) \quad (5)$$

where a_n is the normalized key access age ($0 \leq a_n \leq 1$);

a_{max} is the maximum access age among keys (seconds).

Values closer to 1 indicate that a key has not been accessed for a longer period, thereby assigning it a higher deletion priority.

The next step is to calculate a combined deletion metric (S) that integrates three key parameters: normalized access age, normalized access frequency, and the memory occupied by the key. This metric enables a multicriteria analysis for cache eviction, balancing data retention with efficient memory utilization.

$$S = w_a \times \frac{1}{a_n + \varepsilon} + w_f \times f_n + w_m \times \frac{m_{max}}{m} \quad (6)$$

where S is the deletion score;

w_a is the weight of access age ($0 \leq w_a \leq 1$);

w_f is the weight of access frequency ($0 \leq w_f \leq 1$);

w_m is the weight of memory ($0 \leq w_m \leq 1$);

m is the amount of memory used by the key (bytes);

Рисунок Е.2 – Друга сторінка статті (рисунок виконаний самостійно)

m_{max} is the maximum amount of memory among all keys (bytes);

ε is a small value that prevents division by values in the vicinity of 0.

The inverted normalized access age prioritizes keys that have not been accessed recently, f_n contributes in direct proportion to access frequency, and the memory ratio (maximum memory divided by the key's memory) assigns a higher deletion priority to keys consuming more memory. The equation (6) ensures that keys with older accesses and higher memory usage are more likely to be evicted, while frequently accessed keys are retained.

Weighting factors must sum to 1, as in (7), to balance each metric's influence on the overall S score. Without this constraint, one metric may disproportionately affect the outcome.

$$w_a + w_f + w_m = 1 \quad (7)$$

The values of w_a, w_f, and w_m are set according to system priorities. For instance, if data relevance is most important, a higher w_a is used; if popularity is key, w_f is increased. These coefficients should be determined experimentally – for example, $w_a = 0.2$, $w_f = 0.7$, $w_m = 0.1$ for high request frequency but low relevance, or $w_a = 0.6$, $w_f = 0.3$, $w_m = 0.1$ when data freshness is critical.

The S metric balances memory efficiency and data preservation by integrating access age, frequency, and object size, evicting only those items with the lowest scores to maintain optimal cache performance under varying conditions.

V. Basic Experiment Planning and Design

As part of this research, a utility was developed aimed at improving the efficiency of cached data management in Redis. The main goal of the development is to implement dynamic data lifetime management (TTL) and adaptive cache optimization based on the LRU and LFU algorithms. This approach allows not only to optimize memory usage, but also to maintain stable system performance even under high loads. To achieve these goals, the software was implemented on the .NET platform using the C# programming language and the StackExchange.Redis library, which provides convenient integration with Redis.

A. Experimental Environment

To conduct the experiment, a test environment was created that is as close as possible to the real-world conditions of Redis under high load. The environment is deployed in Docker on a device with an Intel Core i7-12700H, 32 GB of RAM, and an SSD, and the software is developed in .NET (C#) using StackExchange.Redis. The settings include maxmemory and replacement policies (allkeys-lru, allkeys-lfu, allkeys-random, noeviction).

The input data was synthetically generated objects (~3 KB) with Id, Name, Timestamp, and Payload fields (~3000 characters) with TTL from 60 to 120 seconds. Keys were created cyclically, and data was divided into “hot” (20% of keys, 80% of requests) and “cold”, with a variation of the ratio 80:20 and 50:50 to model different load scenarios.

Each test was performed three times with the results averaged with the calculation of the standard deviation to increase reliability. Six levels of the dataset were considered: 10,000, 20,000, 30,000, 50,000, 80,000, and 100,000 records, which allowed us to identify patterns of changes in caching performance. Comprehensive metrics were collected: Cache Hit/Miss Rate, operation time, average TTL, number of evictions, as well as data on memory usage via INFO memory/MEMORY USAGE, CPU, and RAM utilization.

B. Cache Hit and Miss Rates

Fig. 1 and Fig. 2 show the dependences of experimentally measured Cache Hit and Miss Rates, respectively.

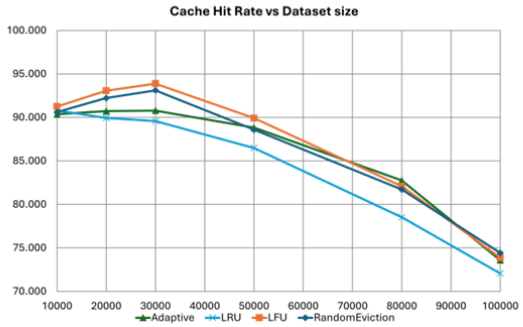


Fig. 1. Charts of cache hit rate vs. dataset size

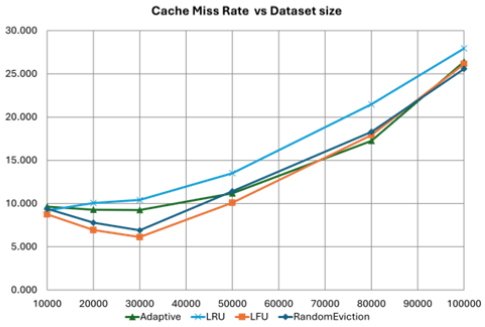


Fig. 2. Charts of cache miss rate vs. dataset size

At 10,000 records, the Cache Hit Rate is high for all strategies (Adaptive – 90.36%, LRU – 90.80%, LFU – 91.25%, Random Eviction – 90.61%) with a low standard deviation (2.03 – 3.75%), indicating stable caching of small amounts of data. At 20,000 records, Adaptive’s performance increases to 90.71%, LFU's to 93.06%, and LRU and Random Eviction show slight fluctuations (89.93% and 92.22%, respectively), with the standard deviation for Adaptive dropping to 1.38%. At 30,000 records, Adaptive remains stable (90.78%), LFU reaches 93.88%, Random Eviction reaches 93.11%, and LRU falls behind to 89.58%, with a very small standard deviation (≈0.27 – 0.28%) for Adaptive. At 50,000 records, all algorithms face a decrease in Hit Rate (Adaptive – 88.82%, LRU – 86.49%, LFU

Рисунок Е.3 – Третя сторінка статті (рисунок виконаний самостійно)

– 89.91%, Random Eviction – 88.59%) with an increase in variability. For 80,000 records, the indicators fall even further (Adaptive – 82.75%, LFU – 82.09%, Random Eviction – 81.71%, LRU – 78.52%) with a standard deviation of 3.69 – 4.24%, and at 100,000 records, the Hit Rate is the lowest (Adaptive – 73.59%, LRU – 72.06%, LFU – 73.86%, Random Eviction – 74.42%) with a deviation of 4.46-5.03%.

C. Evictions

At 10,000 records, the number of evictions is almost the same (Adaptive – 2261, LRU – 2452, LFU – 2403, Random Eviction – 2455), but the variability of Adaptive is the lowest (≈28.88). At 20,000 records, the evisceration rates increase (Adaptive – 4211, LRU – 4458, Random Eviction – 4003), and the variability of Adaptive (≈285.00) is similar to Random Eviction, while LFU has slightly less (≈250.99). At 30,000 records, Adaptive shows 6209 evictions with minimal variability (≈168.54), while LRU shows 6399, LFU shows 5113, and Random Eviction shows 5344, but with more irregularity (≈420.69 and 346.68). At 50,000 records, the performance increases (Adaptive – 8248, LRU –9754, LFU – 7803, Random Eviction – 8076), with Adaptive remaining stable (variability ≈402.24). At 80,000 records, Adaptive has 10979 evictions (variability ≈283.45), almost on par with LFU (10944) and Random Eviction (11195), while LRU is significantly higher (13803). Finally, at 100,000 records, Adaptive is 12697, LFU is 12776, Random Eviction is 13165, and LRU is 16548, with Adaptive's variability (≈348.33) remaining high but lower than LRU (≈310.77) and LFU (≈325.76) (Fig. 3).

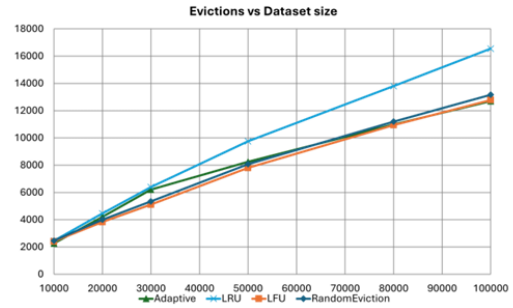


Fig. 3. Charts of evictions vs. dataset size

D. Average TTL

On a set of 10,000 records, Adaptive demonstrates the highest average TTL of 82.26 s (standard deviation ≈0.87 s), outperforming LRU (56.53 s, ≈1.57 s), LFU (54.01 s, ≈0.35 s), and Random Eviction (55.54 s, ≈0.39 s). At 20,000 records, Adaptive slightly increases to 85.08 s (≈2.54 s), while the other strategies fall (LRU – 55.50 s, LFU - 49.78 s, Random Eviction – 51.15 s). For 30,000 records, Adaptive remains at 85.04 s (≈2.52 s), outperforming LRU (53.15 s), LFU (46.64 s), and Random Eviction (47.83 s). At 50,000 writes, Adaptive maintains 85.36 seconds (≈0.57 seconds), compared to LRU's 49.31 seconds, LFU's 43.40 seconds, and Random Eviction's 44.29 seconds. At 80,000 records, Adaptive is 84.29 s (≈0.36 s) compared to LRU 51.76 s, LFU 46.36 s, and Random Eviction 46.85 s. Finally, at 100,000 records, Adaptive remains competitive at 84.75 s (≈0.55 s), while LRU is 55.58 s, LFU is 51.87 s, and Random Eviction is 51.72 s (Fig. 4).

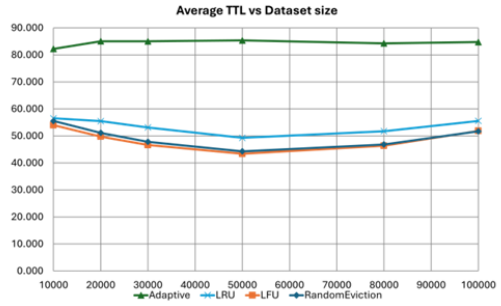


Fig. 4. Charts of average TTL vs. dataset size

E. Average Write and Read Times

Dependences of the average Write and Read times on dataset sizes are shown in Fig.5 and Fig. 6. respectively.

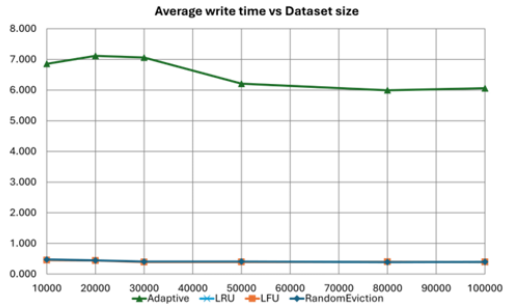


Fig. 5. Charts of average write time vs. dataset size

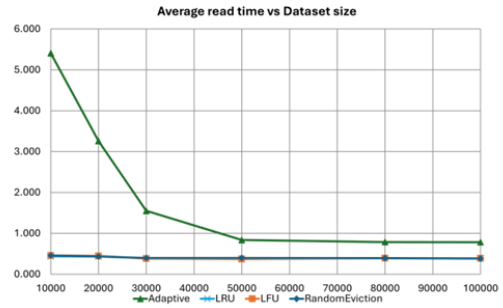


Fig. 6. Charts of average read time vs. dataset size

On a 10,000-entry set, Adaptive has a write time of 6.855 ms/key and a read time of 5.408 ms/key – significantly higher than LRU (0.452/0.436 ms), LFU (0.455/0.459 ms), and Random Eviction (0.475/0.458 ms). At 20,000-30,000 writes, Adaptive's write time remains around 7 ms/key, and read time

Рисунок Е.4 – Четверта сторінка статті (рисунок виконаний самостійно)

drops to 1,550 ms/key, while traditional algorithms are stable (≈0.38-0.46 ms/key). At 50,000 records, Adaptive shows 6.206 ms/key for writing and 0.836 ms/key for reading, and at 80,000 records, 5.993 ms/key and 0.786 ms/key, respectively. Finally, for 100,000 writes, Adaptive achieves the lowest read time of 0.783 ms/key (almost 7 times lower than at 10,000), but the write time remains high at 6.059 ms/key, while LRU, LFU, and Random Eviction are stable at ≈0.39 ms/key for both operations.

F. Memory Usage Delta

At 10,000 records, Adaptive demonstrates a memory usage delta of –4292 bytes with stable cache flushing, while LRU shows a slight fluctuation (+757 bytes) and LFU – +1504 bytes. Random Eviction varies from –779 to +1787 bytes. When the dataset is increased to 100,000 records, Adaptive frees up more memory (–11368 bytes), LRU remains stable (+1219 bytes), and LFU stabilizes at +651 bytes. In general, Adaptive demonstrates active, though not monotonous, memory-freeing dynamics, while LRU provides stable usage, and LFU and Random Eviction are characterized by significant variability depending on the data size (Fig. 7).

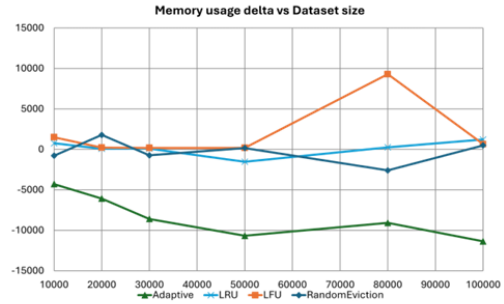


Fig. 7. Charts of memory usage delta vs. dataset size

G. Process CPU Usage

Fig. 8. shows the CPU usage level during the conducted experiment.

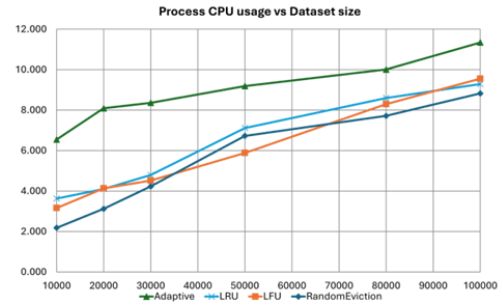


Fig. 8. Charts of process CPU usage vs. dataset size

Adaptive uses a CPU of 6.55% (10,000 keys) to 11.35% (100,000 keys) with high variability on small sets (±2.31%), which decreases to ±0.57% at 100,000 keys. LRU shows a more uniform increase – from 3.63% to 9.29% – with a standard deviation of 1.04-1.26% on small sets, which decreases to ±0.63-0.69% on large sets. LFU starts at 3.16% and reaches 9.55% with 100,000 keys; its variability ranges from 0.47-0.83% to ±1.17%. Random Eviction shows the lowest CPU – from 2.18% to 8.83% with variability falling from ±1.37% (10,000 keys) to an almost stable ±0.07% (100,000 keys).

H. Process Memory Delta

At 10,000 keys, Adaptive demonstrates a Memory Delta of ≈5,312,512 bytes with a high standard deviation (≈2.1×10^6 bytes), which increases to ≈25,643,691 bytes at 100,000 keys (deviation ≈3.2×10^6 bytes). The LRU shows large fluctuations – from ≈2,222,763 bytes at 10,000 to unstable values (an optimum of 845,141 bytes at 50,000 and a sharp increase to ≈14,792,021 bytes at 100,000), with a standard deviation ranging from 161,485 bytes to 10.6×10^6 bytes. LFU is more stable, ranging from ≈3,896,661 bytes to ≈8,844,629 bytes with a standard deviation of 1.4-2.5×10^6 bytes. Random Eviction has the lowest memory consumption, ranging from ≈3,831,125 bytes at 10,000 to ≈7,201,156 bytes at 100,000, with a lower standard deviation (252,719 bytes at 20,000), indicating more predictable memory usage (Fig. 9).

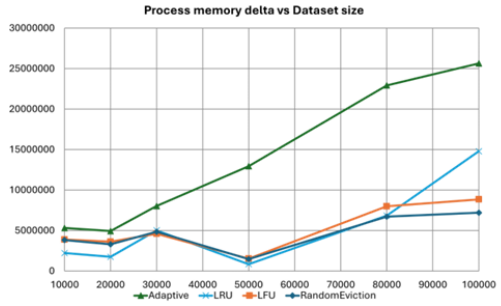


Fig. 9. Charts of process memory delta vs. dataset size

VI. Discussion and Conclusion

The experimental research comparing Adaptive, LRU, LFU, and Random Eviction cache replacement strategies has provided valuable insights into the trade-offs between performance, resource consumption, and cache hit reliability. For small to medium-sized datasets (10,000 – 30,000 records), both LFU and the proposed Adaptive strategy achieve high cache hit rates and maintain stable operation with a superior average TTL. In these scenarios, the Adaptive method's element-level lifetime management shows clear benefits, as it preserves frequently accessed data while efficiently evicting less relevant keys.

As the dataset size increases (50,000 – 100,000 records), all strategies begin to reveal inherent limitations. However, the Adaptive and LFU strategies remain competitive in preserving cache performance. In contrast, LRU, which relies solely on the most recent access time, struggles to maintain efficiency, leading to higher eviction rates and reduced hit ratios. The Adaptive method's ability to dynamically adjust TTL based on both access frequency and recency is especially advantageous

Рисунок Е.5 – П'ята сторінка статті (рисунок виконаний самостійно)

under high load, despite its higher overhead. Notably, while traditional strategies such as LRU and Random Eviction exhibit lower CPU and write-time demands, the Adaptive strategy significantly reduces read time when handling larger datasets – a critical factor in performance-sensitive applications.

Based on the findings of this study, the following focused recommendations are proposed for further research and system optimization:

- for small to medium-sized systems, both Adaptive and LFU strategies provide optimal cache performance. However, when minimizing write time and CPU usage is crucial, LRU or Random Eviction may be preferred;
- future works should focus on reducing the computational overhead of the Adaptive strategy. Techniques such as further algorithmic optimizations or hardware-specific tuning could help achieve a more balanced trade-off between resource consumption and read efficiency;
- future works could explore integrated approaches that leverage real-time system monitoring to dynamically select or blend Adaptive, LFU, and LRU techniques. This strategy would serve as a complementary enhancement to the current study, optimizing cache performance under varying workload conditions;
- experimentation with weighting factors (w_a, w_f, w_m) in the deletion metric and adjustment of TTL boundaries (T_{base} and T_{max}) can further refine performance. Tailoring these parameters to specific application profiles may yield additional gains in memory utilization and response times.

In conclusion, the research validates the promise of adaptive cache management in environments with complex data structures and variable load conditions. While the Adaptive approach incurs higher initial overhead, its significant reduction in read latency and its effective balancing of eviction priorities justify its use in high-load, performance-critical applications. Continued refinement and the integration of hybrid models stand as perspective directions for future research, with the goal of developing more robust and efficient caching solutions for modern IT systems.

REFERENCES

[1] V. Guzhov and K. Smelyakov, "Free DBMSs Performance for an Inventory Management System based on Spring Boot," in *2024 IEEE Open Conf. of Electrical, Electronic and Information Sciences (eStream)*, Vilnius, Lithuania, 2024, pp. 1-5, doi: 10.1109/eStream61684.2024.10542597

[2] M. Ç. Kara, M. Elamine Benlakehal, I. Shayea, A. Tussupov and L. Rzayeva, "Data Caching in Edge Computing: A Survey," in *2024 IEEE 4th International Conf. on Smart Information Systems and Technologies (SIST)*, Astana, Kazakhstan, 2024, pp. 433-439, doi:10.1109/SIST61555.2024.10629324

[3] A. Kuzochkina, M. Shirokopetleva and Z. Dudar, "Analyzing and Comparison of NoSQL DBMS," in *2018 International Scientific-Practical Conf. Problems of Infocommunications. Science and Technology* (*PIC S&T)*, Kharkiv, Ukraine, 2018, pp. 560-564, doi: 10.1109/INFOCOMMST.2018.8632133

[4] Redis, "What Are the Impacts of the Redis Expiration Algorithm?" 2024. Accessed: Oct. 15, 2024. [Online]. Available: https://redis.io/kb/doc/1fqjridk8w/what-are-the-impacts-of-the-redis-expiration-algorithm

[5] S. Basu, A. Sundarrajan, J. Ghaderi, S. Shakkottai and R. Sitaraman, "Adaptive TTL-Based Caching for Content Delivery," *IEEE/ACM Transactions on Networking*, vol. 26, no. 3, pp. 1063-1077, June 2018, doi: 10.1109/TNET.2018.2818468

[6] J. Liu, X. Wan, Q. Zhu, T. Peng and X. Hu, "Research on Adaptive Cache Mechanism Based on TTL," in *2022 2nd International Conf. on Networking, Communications and Information Technology (NetCIT)*, Manchester, United Kingdom, 2022, pp. 507-511, doi: https://doi.org/10.1109/NetCIT57419.2022.00125

[7] P. Zhang, L. Xing, N. Yang, G. Tan, Q. Liu and C. Zhang, "Redis++: A High Performance In-Memory Database Based on Segmented Memory Management and Two-Level Hash Index," in *2018 IEEE Intl Conf. on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, Melbourne, VIC, Australia, 2018, pp. 840-847, doi: 10.1109/BDCloud.2018.00125

[8] Q. Zheng, T. Yang, Y. Kan, X. Tan, J. Yang and X. Jiang, "On the Analysis of Cache Invalidation With LRU Replacement," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 654-666, 1 March 2022, doi: 10.1109/TPDS.2021.3098459

[9] Z. Shen, B. Jiang, X. Wang and C. Zhou, "ARC-learning: A Self-tuning Cache Policy under Dynamic Popularity," in *2022 IEEE 8th International Conf. on Computer and Communications (ICCC)*, Chengdu, China, 2022, pp. 610-615, doi: 10.1109/ICCC56324.2022.10065823

[10] L. Ait-Oucheggou, S. Rubini, A. Battou and J. Boukhobza, "Investigating Multi-Tier and QoS-Aware Caching Based on ARC," *2023 31st International Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Stony Brook, NY, USA, 2023, pp. 1-4, doi: 10.1109/MASCOTS59514.2023.10387601

[11] M. Burhan, A. Arsalan and R. A. Rehman, "EPP-LFU: An Efficient Producer Popularity-based LFU Policy for the Applications of Named-Data Network," in *2022 24th International Multitopic Conf. (INMIC)*, Islamabad, Pakistan, 2022, pp. 1-6, doi: 10.1109/INMIC56986.2022.9972919

[12] K. Smelyakov, A. Chupryna, D. Darahan and S. Midina, "Effectiveness of modern text recognition solutions and tools for common data sources," in *Proc. of the 5th International Conf. on Computational Linguistics and Intelligent Systems (COLINS 2021)*, vol. I (2870)*:* Main Conference, Lviv, Ukraine, April 22-23, 2021, pp. 154-165, [Online]. Available: https://ceur-ws.org/Vol-2870

[13] S. Das and A. Banerjee, "An arbitration on cache replacements based on frequency – Recency product values," in *2016 International Conf. on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)*, Bengaluru, India, 2016, pp. 1-6, doi: 10.1109/VLSI-SATA.2016.7593031

[14] C. Cho, S. Shin, H. Jeon and S. Yoon, "TTL-Based Cache Utility Maximization Using Deep Reinforcement Learning," in *2021 IEEE Global Communications Conf. (GLOBECOM)*, Madrid, Spain, 2021, pp. 1-6, doi: 10.1109/GLOBECOM46510.2021.9685845

[15] D. Carra, G. Neglia and P. Michiardi, "TTL-based Cloud Caches," *IEEE INFOCOM 2019 - IEEE Conf. on Computer Communications*, Paris, France, 2019, pp. 685-693, doi: 10.1109/INFOCOM.2019.8737546

[16] J. Shah and A. A. Siddiqui, "An Improved Cache Eviction Strategy: Combining Least Recently Used and Least Frequently Used Policies," in *2023 6th International Conf. on Advances in Science and Technology (ICAST)*, Mumbai, India, 2023, pp. 1-6, doi: 10.1109/ICAST59062.2023.10454976

Рисунок Е.6 – Шоста сторінка статті (рисунок виконаний самостійно)

ДОДАТОК Ж

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-23-1 (група)

Кучапін Матвій Юрійович
(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 познаки	
	7.15 Додатки	

зауважень немає

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

29.05.2025

Рисунок Ж.1 – Експертний висновок результатів перевірки кваліфікаційної роботи (рисунок виконаний самостійно)