А не зашкалит сервер от 100 000 запросов/сек

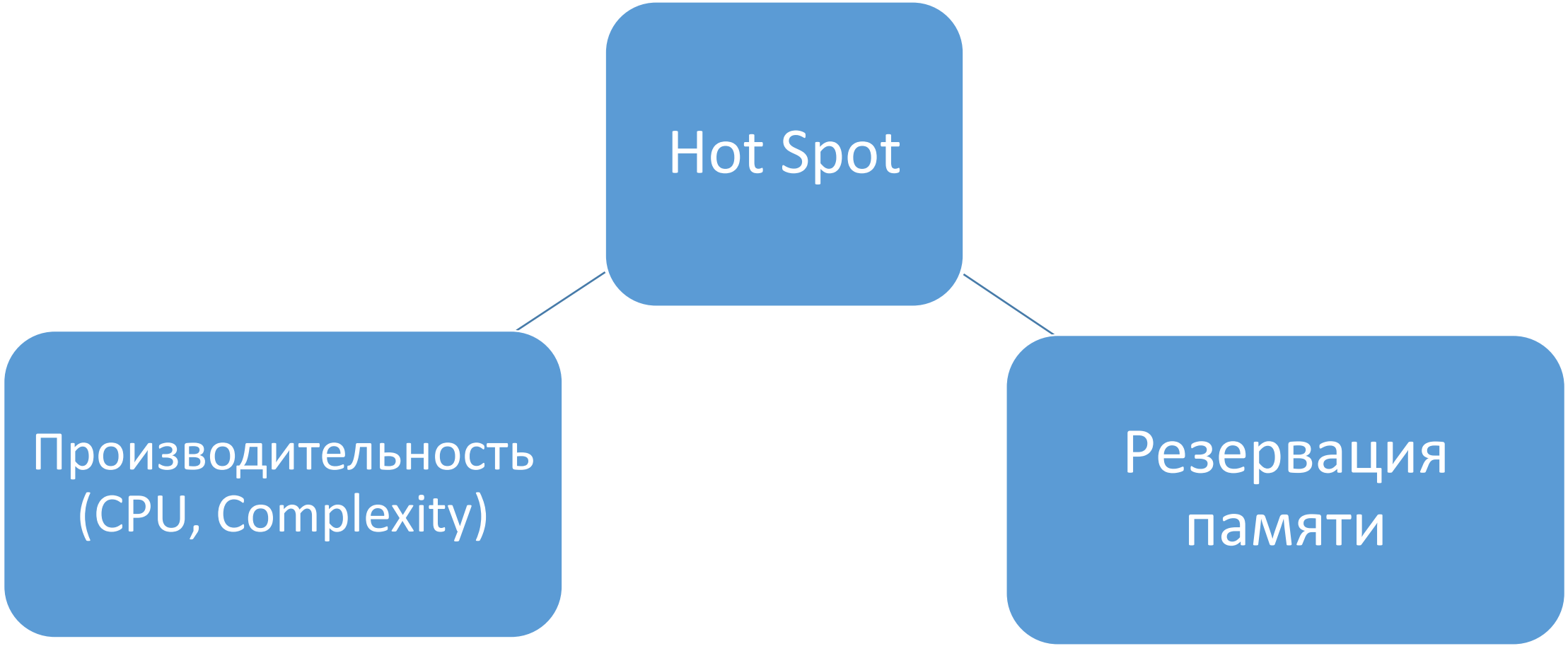
Михаил Ярийчук

Hibernating Rhinos

Алгоритм оптимизации

1. Найти проблемный участок кода (hot spot)
2. Заменить общий алгоритм на специализированный для решения конкретной проблемы

Хорошо, а что такое Hot Spot?



Документноориентированная СУБД

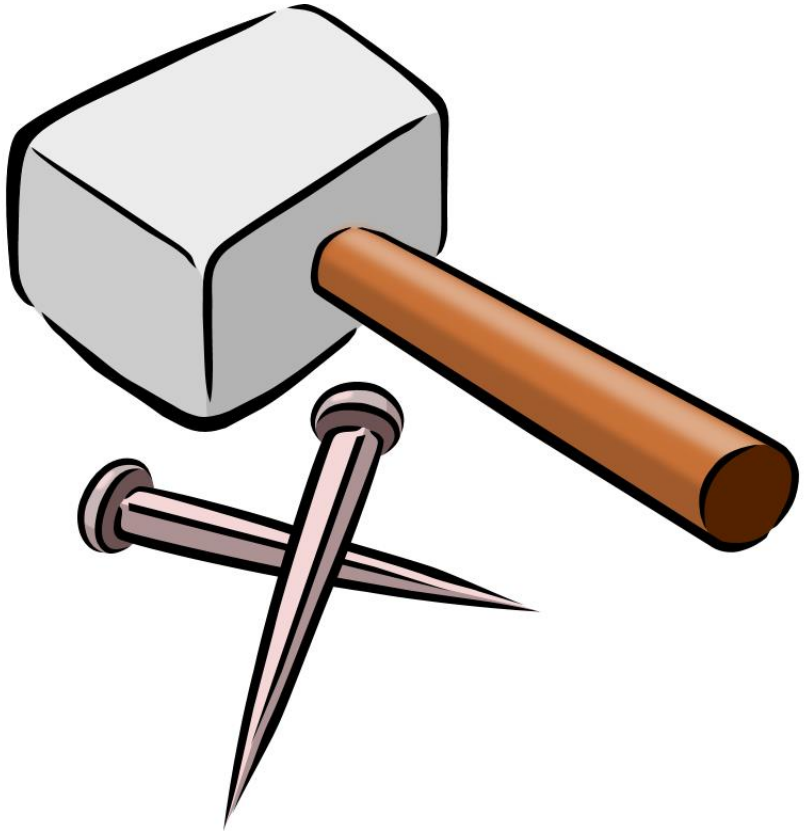
документ

```
{
    "Contact": {
        "Name": "Shelley Burke",
        "Title": "Order Administrator"
    },
    "Name": "New Orleans Cajun Delights",
    "Address": {
        "Line1": "P.O. Box 12345",
        "Line2": null,
        "City": "New Orleans",
        "Region": "LA",
        "PostalCode": "4321",
        "Country": "USA"
    },
    "Phone": "(100) 555-1122",
    "Fax": null,
    "HomePage": "#CAJUN.HTM#"
}
```

RAVENDB

Safe by Default, Optimized for Efficiency

Забивать гвозди микроскопом?



https://openclipart.org/

Мотивация, или с чего начались оптимизации?

- Часто резервируется новая память
- Как следствие много GC
- Esent, существующее хранилище ключ/значение не возможно оптимизировать - закрытый код

<u>Данный сценарий использования</u>

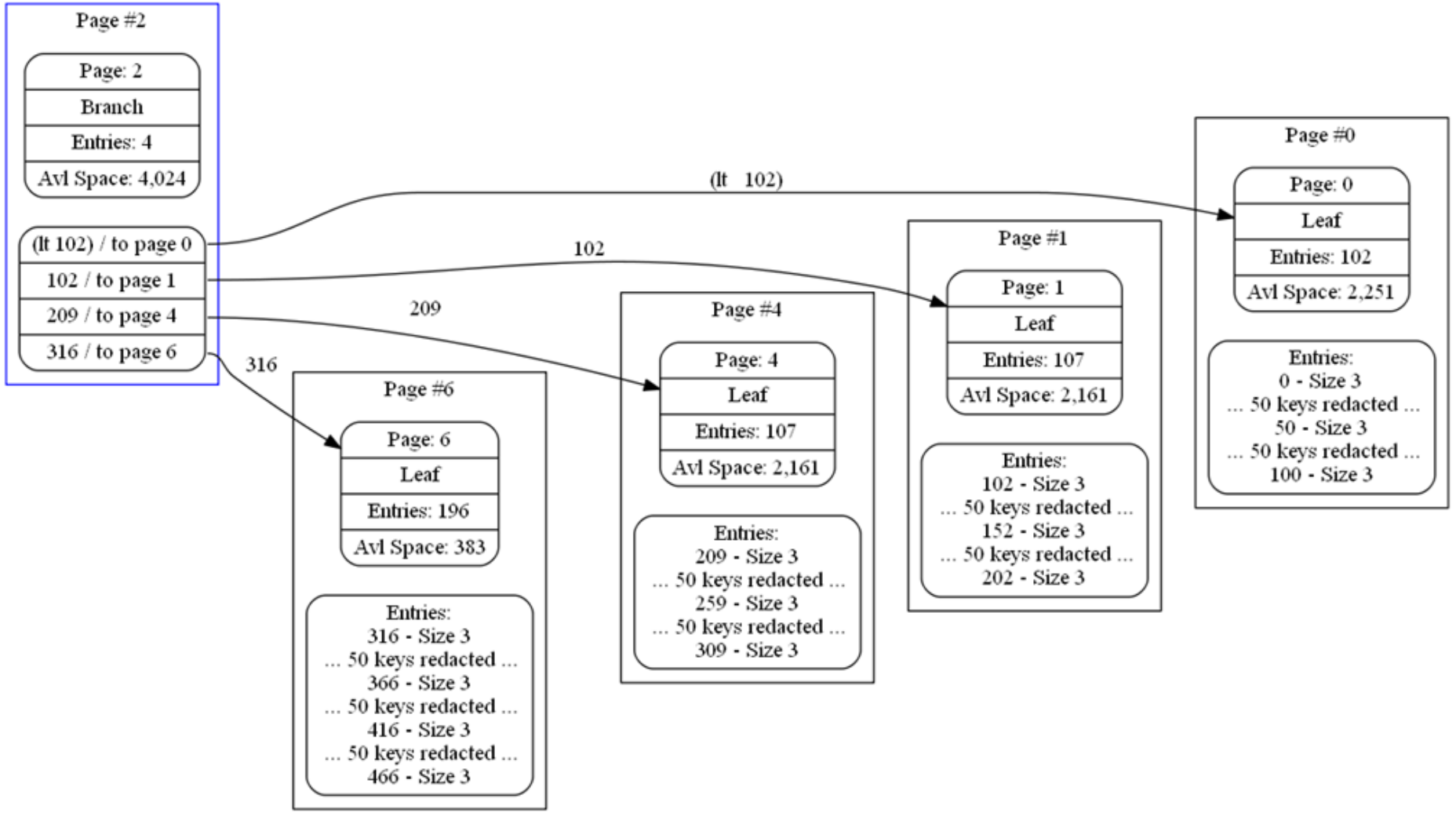
простой Bulk Insert - 4.5 миллиона документов

Voron - хранилище ключ/значение

- Написано на C# для использования в RavenDB
- Использует оперативную память вне .Net Heap
- Функционально аналогичен Esent'y

  (с точки зрения гарантий ACID, транзакционности и принципа сохранения данных)

Немного о дизайне Ворона 1



Ворон и Zero-Copy

```
[DllImport("kernel32.dll", SetLastError = true)]
public static extern byte* MapViewOfFileEx(IntPtr hFileMappingObject,
                                           NativeFileMapAccessType dwDesiredAccess,
                                           uint dwFileOffsetHigh,
                                           uint dwFileOffsetLow,
                                           UIntPtr dwNumberOfBytesToMap,
                                           byte* lpBaseAddress);

[DllImport("kernel32.dll", SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
public static extern bool FlushViewOfFile(byte* lpBaseAddress, IntPtr dwNumberOfBytesToFlush);
```

Немного о дизайне Ворона 2
Резервирование новой ячейки дерева B+

```
var newNodeOffset = (ushort)(header->Upper - nodeSize);

var node = (TreeNodeHeader*)(Base + newNodeOffset);
KeysOffsets[index] = newNodeOffset;

header->Upper = newNodeOffset;
header->Lower += Constants.NodeOffsetSize;
```

Немного о дизайне Ворона 2
Чтение ячейки дерева B+

```
public TreeNodeHeader* GetNode(int n)
{
    return (TreeNodeHeader*)(Base + KeysOffsets[n]);
}
```

Результаты проверок

до оптимизаций (версия 3.5)

- 1 500 записей/сек
- 46 000 запросов/сек, 100% CPU

после оптимизаций (версия 4.0)

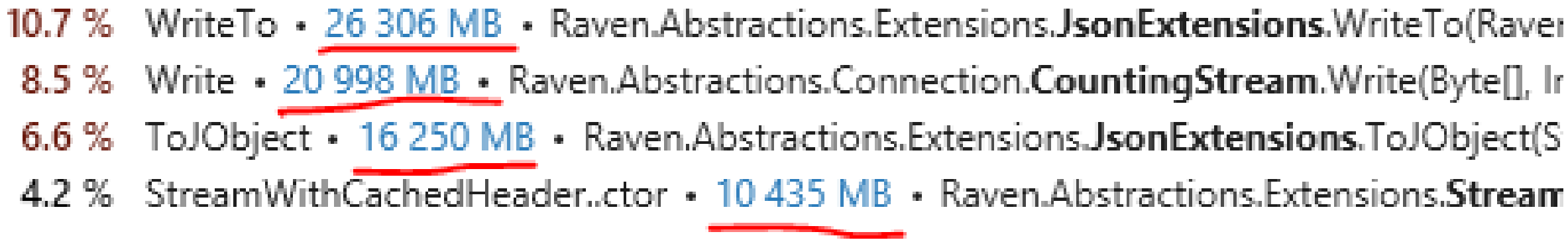
- 53 000 записей/сек*
- 153 000 запросов/сек, 70% CPU*

** предварительный результат проверок*

Неплохо, но недостаточно

Цена сериализации JSON - до оптимизации

- Величина данных : 3GB
- Работают только индексы
- После 10 минут работы

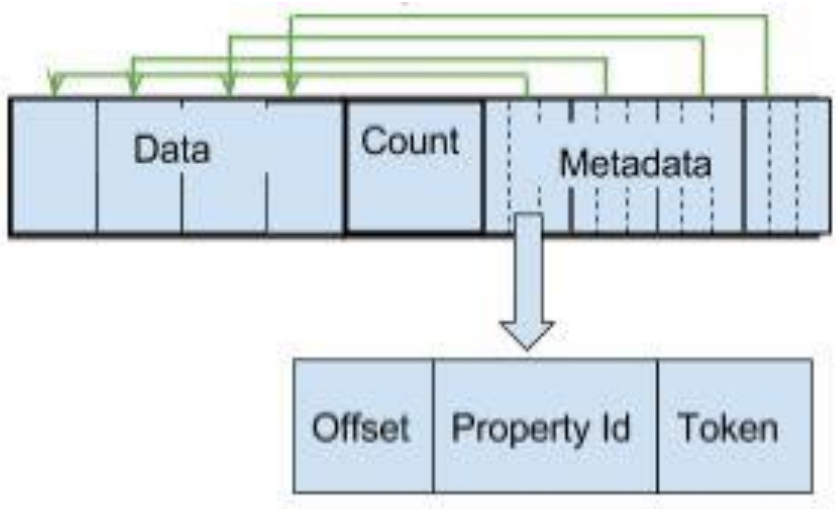


Blittable Json

- Библиотека сериализации/десериализации Json, написана на C#
- Используется в RavenDB 4.0
- Доступ к данным без десериализации и дополнительного выделения памяти
- Исходники здесь: https://github.com/ravendb/ravendb/tree/v4.0/src/Sparrow

Blittable JSON - *исходная величина Json 37KB*

VS



20 bytes

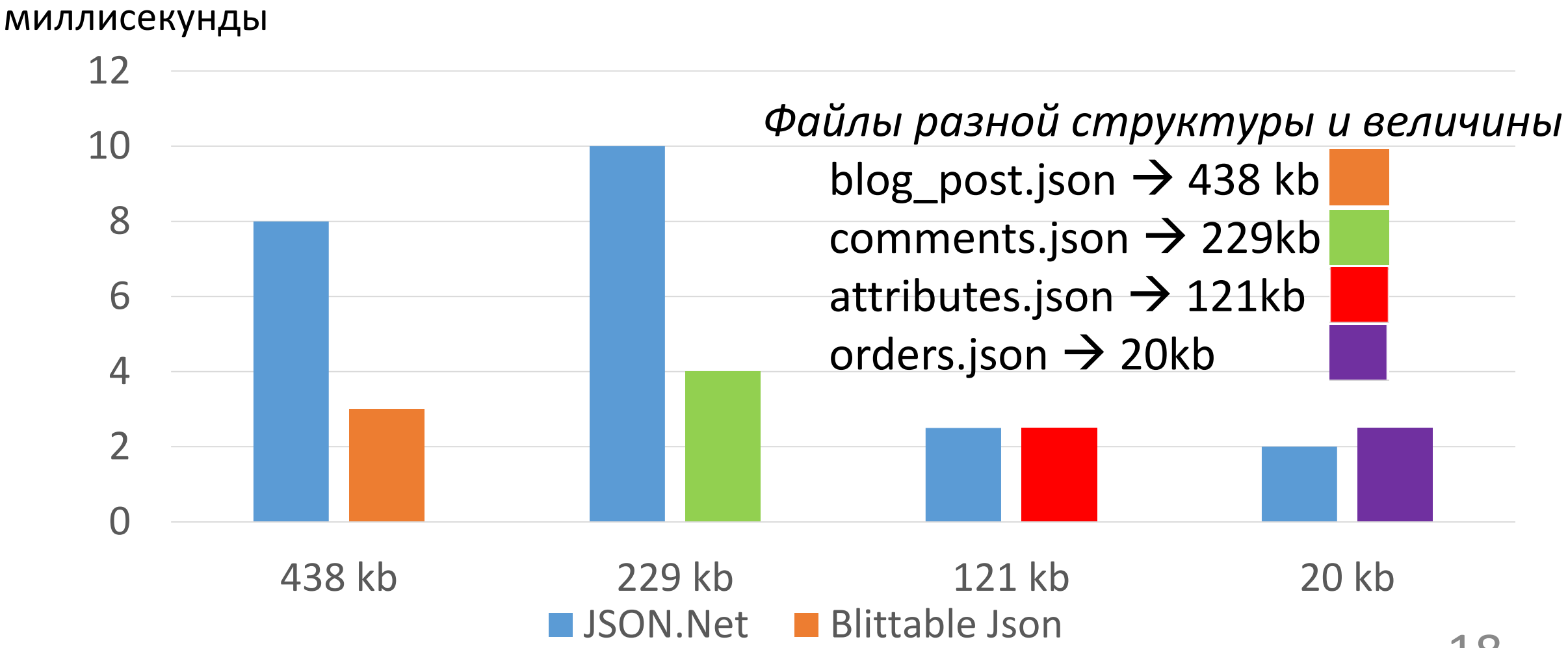
```
private byte* _metadataPtr;
private readonly int _size;
private readonly int _propCount;
private readonly long _currentOffsetSize;
private readonly long _currentPropertyIdSize;
```

Blittable JSON

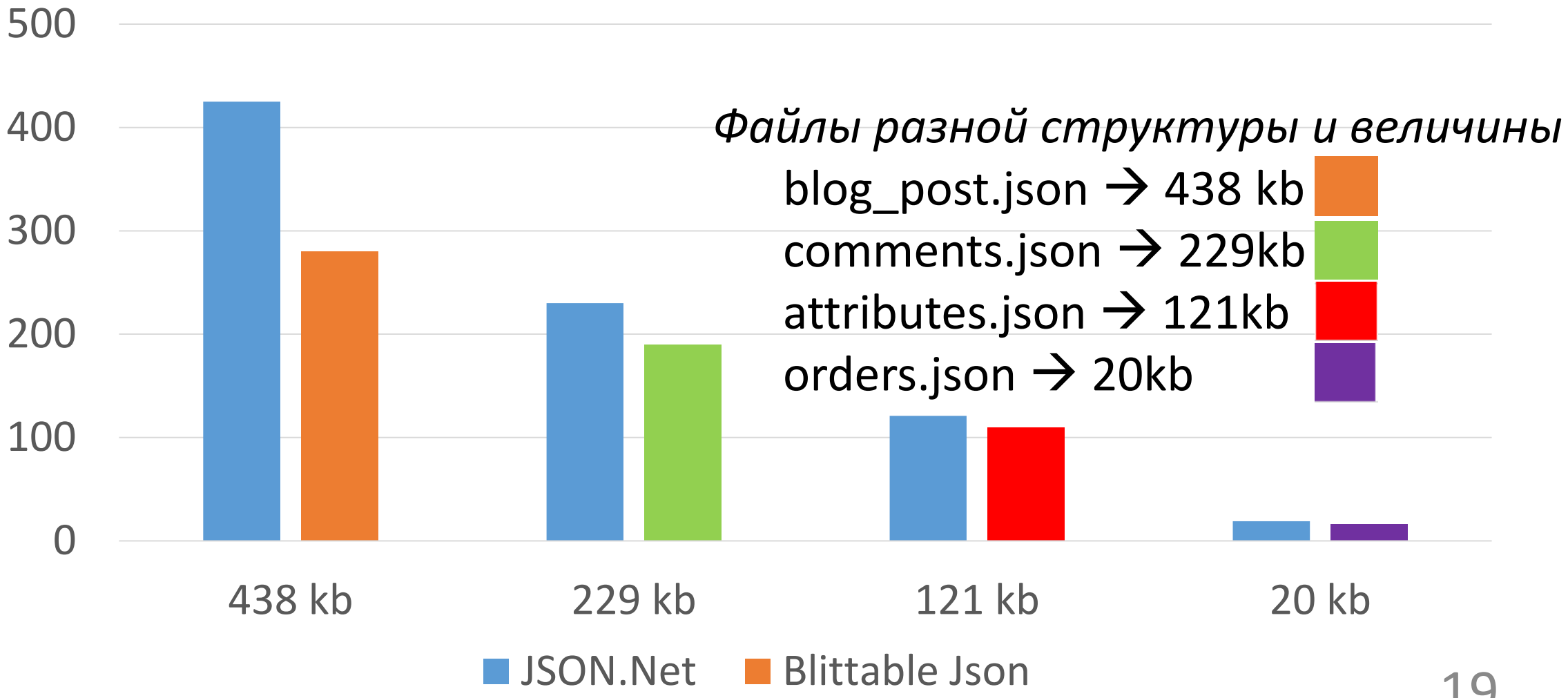
```
public LazyStringValue ReadStringLazily(int pos)
{
    byte offset;
    var size = ReadVariableSizeInt(pos, out offset);

    return new LazyStringValue(null, _mem + pos + offset, size, _context);
}
```

Результат - Скорость



Результат - Величина



Файлы разной структуры и величины

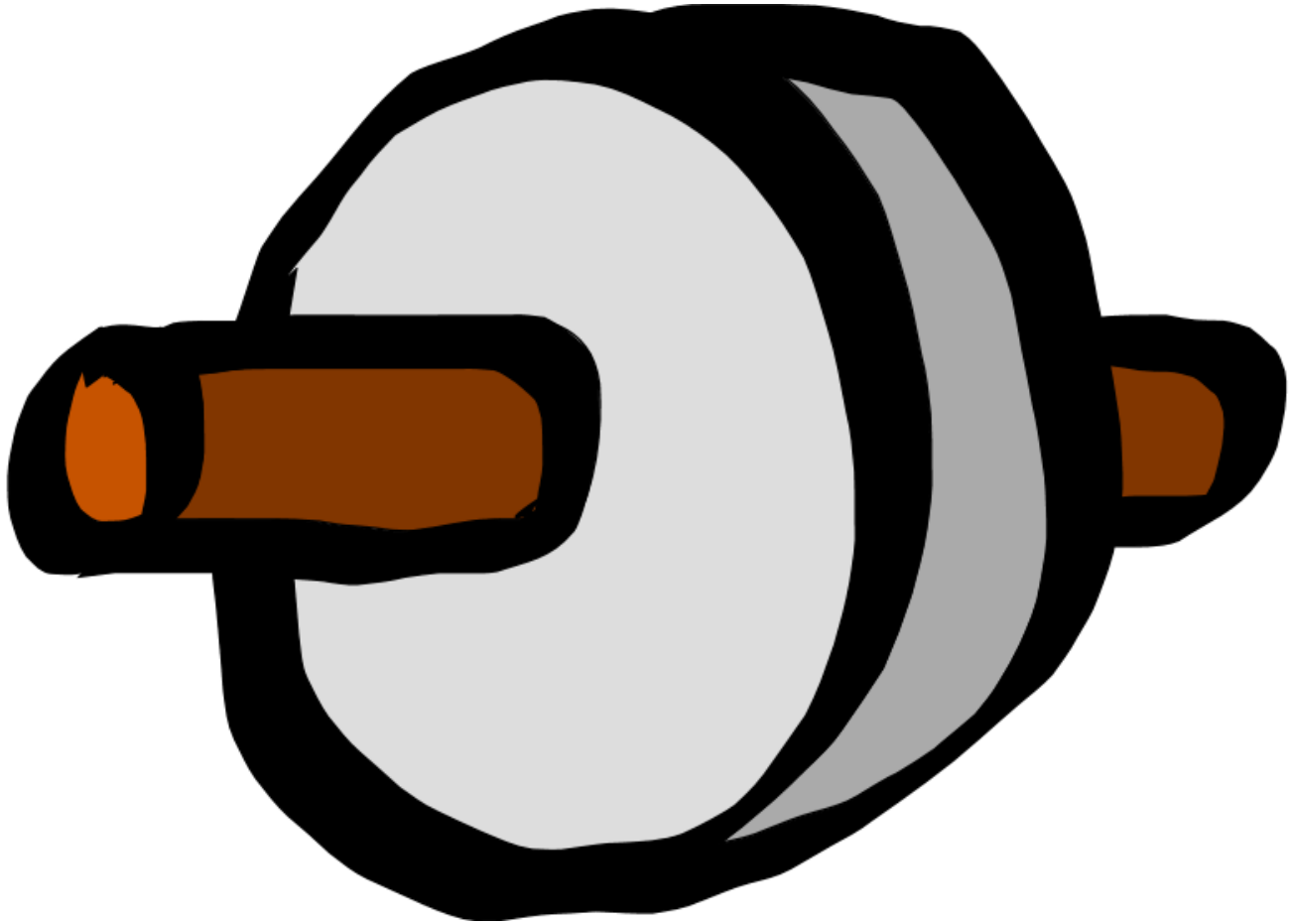
blog_post.json → 438 kb

comments.json → 229kb

attributes.json → 121kb

orders.json → 20kb

Изобретаем колесо?



https://openclipart.org/

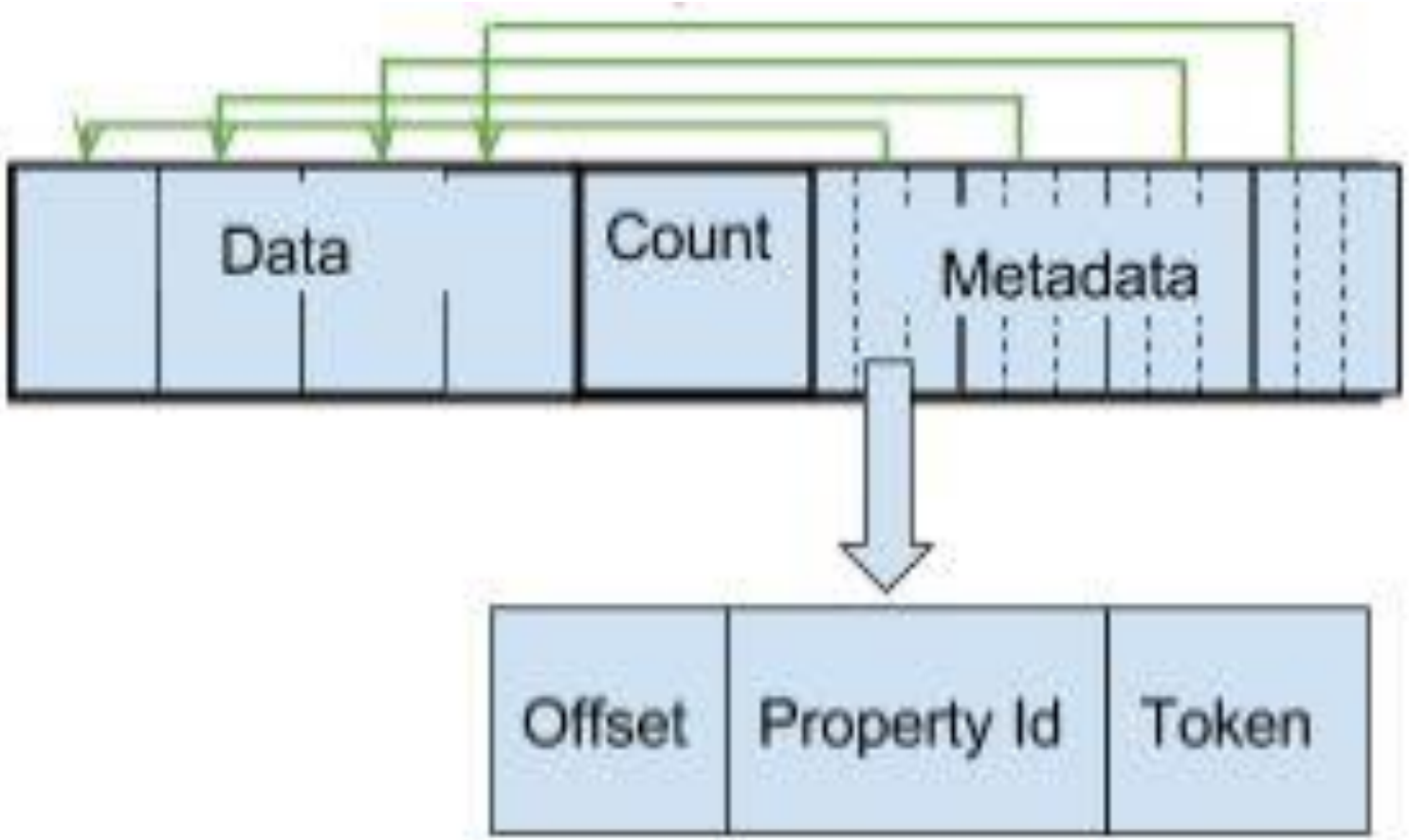
Есть еще место для оптимизаций…

Type	Allocated bytes	Allocated objects	Collected bytes
System.String	11,976,067,918	164,157,269	11,975,260,778

Операционные Контексты

- Контейнер для кэшей, объектов с ограниченным временем жизни и алокаторов
- Контекст выделяется на каждую серверную операцию. Таким образом в конце операции контекст освобождает ресурсы
- Кэш в контекстах сохраняется между операциями

Операционные Контексты
Blittable Json Property Cache



Операционные Контексты
Blittable Json Property Cache

Для Json документа:

```
{
   "FirstName": "John",
   "LastName": "Dow",
   "Age": 30,
   "Birthday": "1980-1-1"
}
```

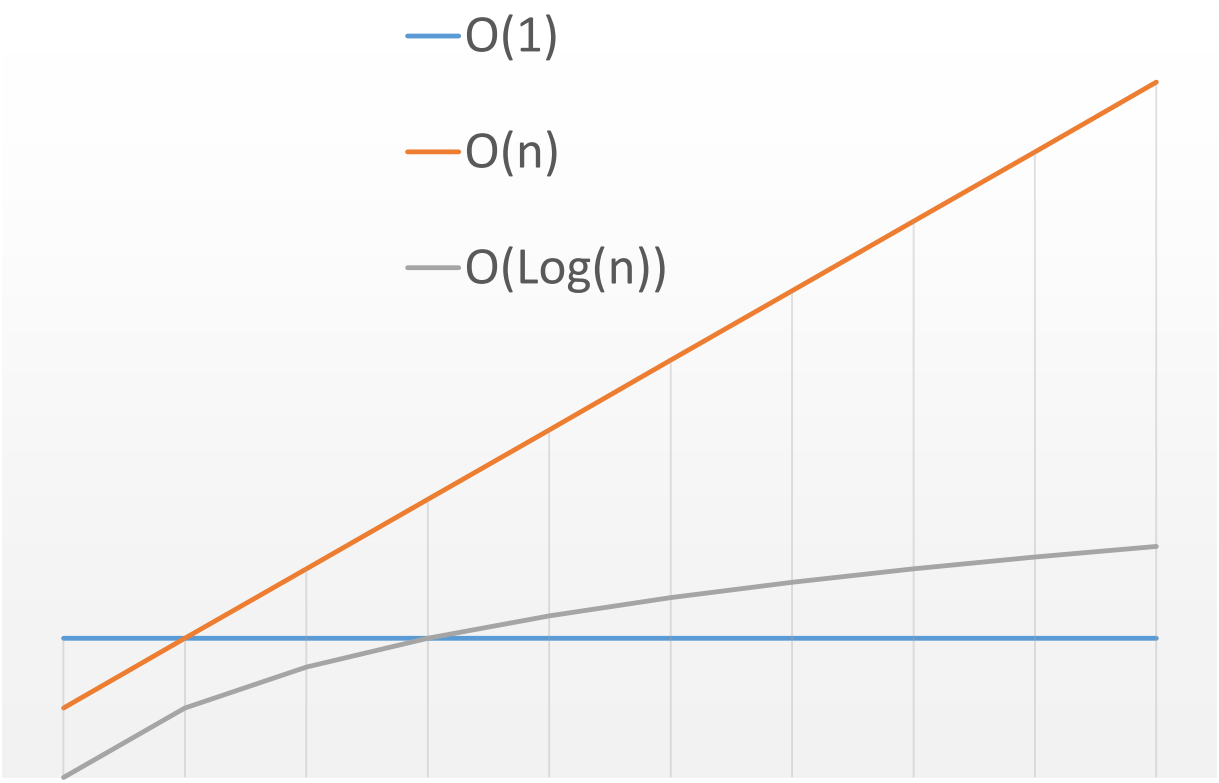
Операционные Контексты
Blittable Json Property Cache

будет вот такая метадата в Blittable Json :

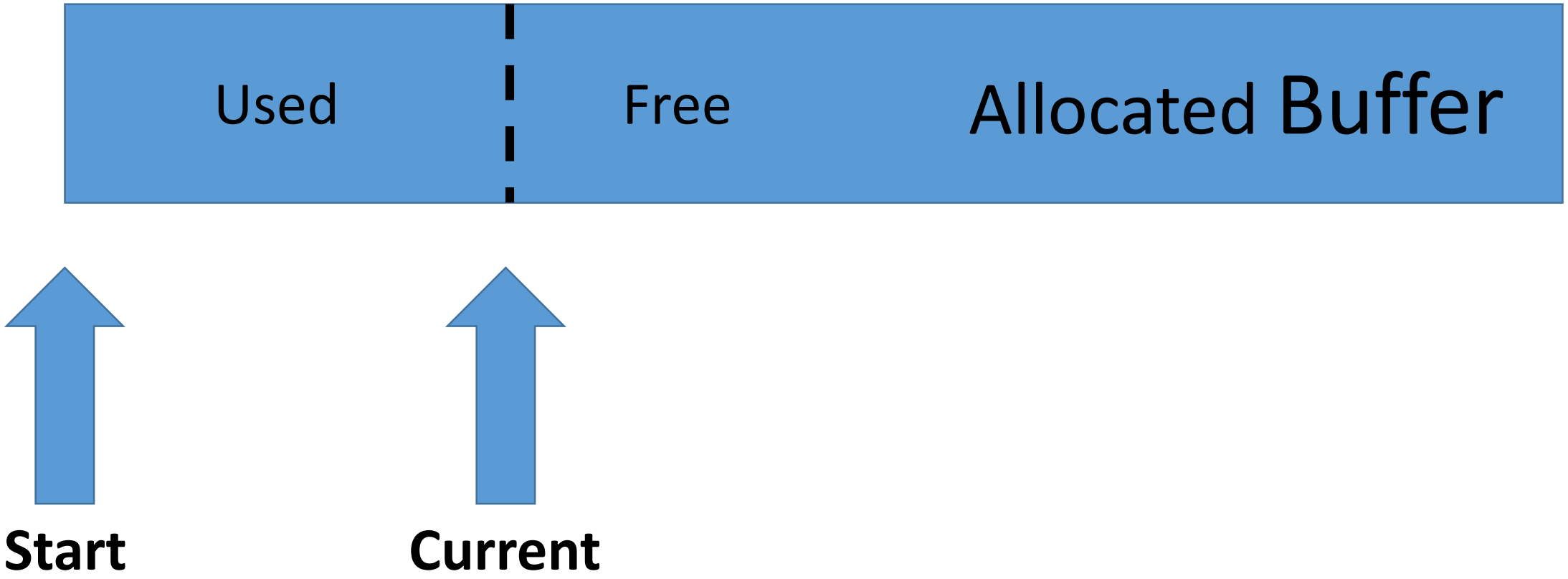
Metadata Entry	Metadata Entry	Metadata Entry	Metadata Entry
• Prop: “Age” • Value Offset	• Prop: “Birthday” • Value Offset	• Prop: “FirstName” • Value Offset	• Prop: “LastName” • Value Offset

Операционные Контексты - Blittable Json Property Cache

- Поиск аттрибута (O(N))
- Доступ к аттрибуту по имени, поиск по метадате → O(log(n))
- Кэш позволяет доступ со сложностью O(1)



Операционные Контексты - Arena Memory Allocator



Операционные Контексты - Arena Memory Allocator

```
public unsafe class ArenaMemoryAllocator : IDisposable
{
    private byte* _ptrStart;
    private byte* _ptrCurrent;

    private long _allocated;
    private long _used;
```

Операционные Контексты - Arena Memory Allocator

```
public AllocatedMemoryData Allocate(int size)
{
    if (_used + size > _allocated)
        GrowArena(size);

    var allocation = new AllocatedMemoryData()
    {
        SizeInBytes = size,
        Address = _ptrCurrent
    };

    _ptrCurrent += size;
    _used += size;

    return allocation;
}
```

Сложность O(1)

Операционные Контексты - ByteString

Использует очень мало Managed Memory..

```
public unsafe struct ByteString : IEquatable<ByteString>
{
    internal ByteStringStorage* _pointer;

    public byte this[int index]
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        get
        {
            return *(_pointer->Ptr + (sizeof(byte) * index));
        }
    }
```

Операционные Контексты - ByteString

- Использует Arena Allocator
- Написанно для уменьшение количества строк в Managed Heap
- Таким образом, меньше циклов GC

Операционные Контексты – ByteString еще одна оптимизация…

```
public void Release(ref ByteString value)
{
     _currentlyAllocated -= value._pointer->Size;
    int reusablePoolIndex = GetPoolIndexForReuse(value._pointer->Size);

    if (value._pointer->Size <= ByteStringContext.MinBlockSizeInBytes)
    {
        Stack<IntPtr> pool = this._internalReusableStringPool[reusablePoolIndex];
        if (pool == null)
        {
            pool = new Stack<IntPtr>();
            this._internalReusableStringPool[reusablePoolIndex] = pool;
        }

        pool.Push(new IntPtr(value._pointer));
        this._internalReusableStringPoolCount[reusablePoolIndex]++;
    }
```

Операционные Контексты - ByteString

```
public struct ByteStringMemoryCache : IByteStringAllocator
{
    public UnmanagedGlobalSegment Allocate(int size)...

    public void Free(UnmanagedGlobalSegment memory)...

    [ThreadStatic]
    private static Stack<UnmanagedGlobalSegment> _threadLocal;
```

Операционные Контексты
Когда это надо?

- Распределения памяти приводят к частым циклам GC - процесс проводит 50% времени в GC и выше
- Специфические объекты вроде строк составляют ощутимую часть объектов в Managed Heap

Сложные оптимизации нужны не всегда

Проблема:

- Процесс адресовки запроса к соответствующей функции обработки берет около 90% запроса
- Запрос проходит через WebAPI, оптимизация невозможна

Простое решение – Prefix Trie

Kestrel дает способ определять свою маршрутизацию

```
public Trie<T> SearchTrie(Trie<T> current, string term)
{
    for (int i = 0; i < term.Length; i++)
    {
        if (CurrentIndex < current.Key.Length)
        {
            if (CharEqualsAt(current.Key, CurrentIndex, term, i) == false)
            {
                if (current.Key[CurrentIndex] == '$')
                {
                    Match.MatchLength = i;
                    Value = current.Value;
                    return current;
                }
                return current;
            }
            CurrentIndex++;
            continue;
        }
```

Сравнительная стоимость маршрутизации после оптимизации

	Процент времени запроса	Среднее время запросов
WebAPI	90%	1 449 194 ms
NancyFX	1.91%	64 740 ms
Custom Routing*	1.02%	2 238 ms

**после оптимизации*

Так о чем все это?

- Узкоспециализированные решения эффективнее чем общие решения
- Простые решения эффективнее сложных

А еще - оптимизации это как микроскоп...

https://openclipart.org/

Benchmarks…

https://openclipart.org/

А теперь немного цифр

Wrk – Http Benchmarking Tool
https://github.com/wg/wrk

Тестовая нагрузка :
256 connections / 8 threads
Работа с базой данных Stack Overflow, общая величина 90GB

./wrk -c 256 -t 8 http://127.0.0.1:8080 -s consistent-questions.lua -- 32

А теперь немного цифр

Использование памяти во время проверки

- Managed memory → 250 MB
- Unmanaged memory → 1 GB

Величина данных на диске → 90GB

А теперь немного цифр

Конкретный пример

- Данные о пользователях сайта Stackoverflow → 5.9 миллионов документов)
- Величина данных около 5GB.
- Map/reduce индекс (агрегация) - количество регистраций пользователей по месяцу

А тепер немного цифр

В среднем, индексация заканчивается меньше чем через минуту

А тепер немного цифр

Тест 1 : Загрузка случайного документа по ID в отдельной транзакции

Средний результат → 153 325,18 запросов / сек

А тепер немного цифр

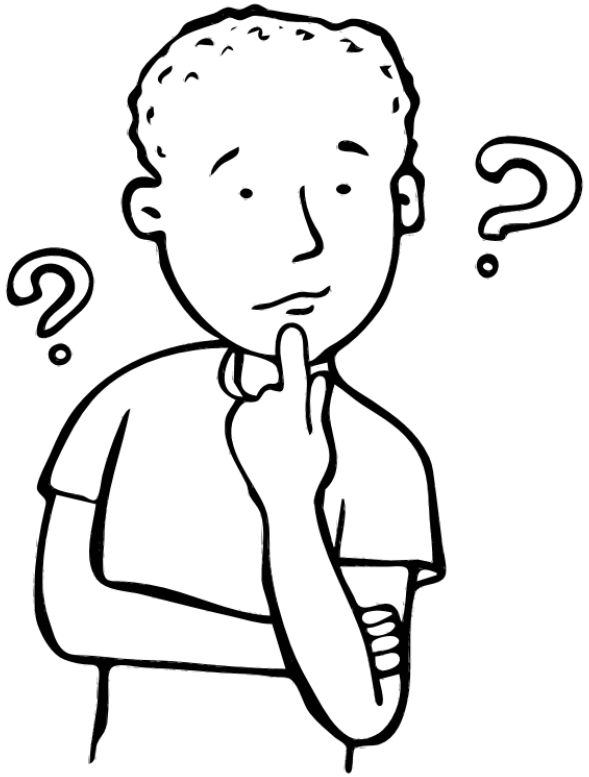
Тест 2 : Сохранение документа в отдельной транзакции.

Средний результат - 53 201 запросов / сек

Вопросы?

michael.yarichuk@hibernatingrhinos.com
@myarichuk

https://github.com/ravendb/ravendb/tree/v4.0



https://openclipart.org/