Efficient & Fast Text Processing

Sebastian Hofstätter

sebastian.hofstaetter@tuwien.ac.at

Today

Efficient & Fast Text Processing

① Measure!
② Garbage collection
③ Memory mapped files
④ Working with UTF-8
⑤ Using the right data structures
⑥ Parallelization

Please ask questions at any point!

Performance

- Performance is language / runtime dependent
  - But overall themes are language agnostic

- Some languages are better suited than others for performance
  - But a lot depends on high-level decisions and not on bit packing tricks

- Not every piece of code has to be optimized
  - But if your code block is being called a billion times for every processed GB 🤷‍♀️ optimize it!

Performance is a feature

- Great example: Stack Overflow https://stackexchange.com/performance
  - **1.3 billion page views / month** with 18ms rendering time
  - 9 web servers (mostly redundant)
  - 2 SQL servers (11.000 queries / second peak load per server)
  - 1 Redis + 3 Elasticsearch + 1 proxy server (+ redundancies)

  - Powered by C# and ASP.NET (statically typed + memory managed runtime)

Details: https://nickcraver.com/blog/2016/03/29/stack-overflow-the-hardware-2016-edition/

And more in-depth: https://blog.marcgravell.com/2016/05/how-i-found-cuda-or-rewriting-tag.html

A few notes …

- The purpose of this lecture is to give you inspirations
  & some practical skills for the exercise and beyond
- You are not required to use anything shown **(except the profiler!)**
- There is **much much** more out there than in this lecture!
- Examples in this lecture are C# (but as much language agnostic as possible)

Computer latencies at human scale

System Event	**Actual Latency**	**Scaled Latency**
One CPU cycle	0.4 ns	**1 s**
Level 1 cache access	0.9 ns	**2 s**
Level 2 cache access	2.8 ns	**7 s**
Level 3 cache access	28 ns	**1 min**
Main memory access (DDR DIMM)	~100 ns	**4 min**
Intel Optane memory access	<10 µs	**7 hrs**
NVMe SSD I/O	~25 µs	**17 hrs**
SSD I/O	50–150 µs	**1.5–4 days**
Rotational disk I/O	1–10 ms	**1–9 months**
Internet call: San Francisco to New York City	65 ms	**5 years**
Internet call: San Francisco to Hong Kong	141 ms	**11 years**

Originally in *Systems Performance: Enterprise and the Cloud, adapted from* *https://www.prowesscorp.com/computer-latency-at-a-human-scale*

Measure!

Using profilers & benchmark libraries

Why measure?

- Modern hardware/OS stack is so complex, that we cannot predict the impact of low-level changes
- Our intuition might be wrong (especially if we work on already optimized code)
- Gives you concrete evidence of improvement

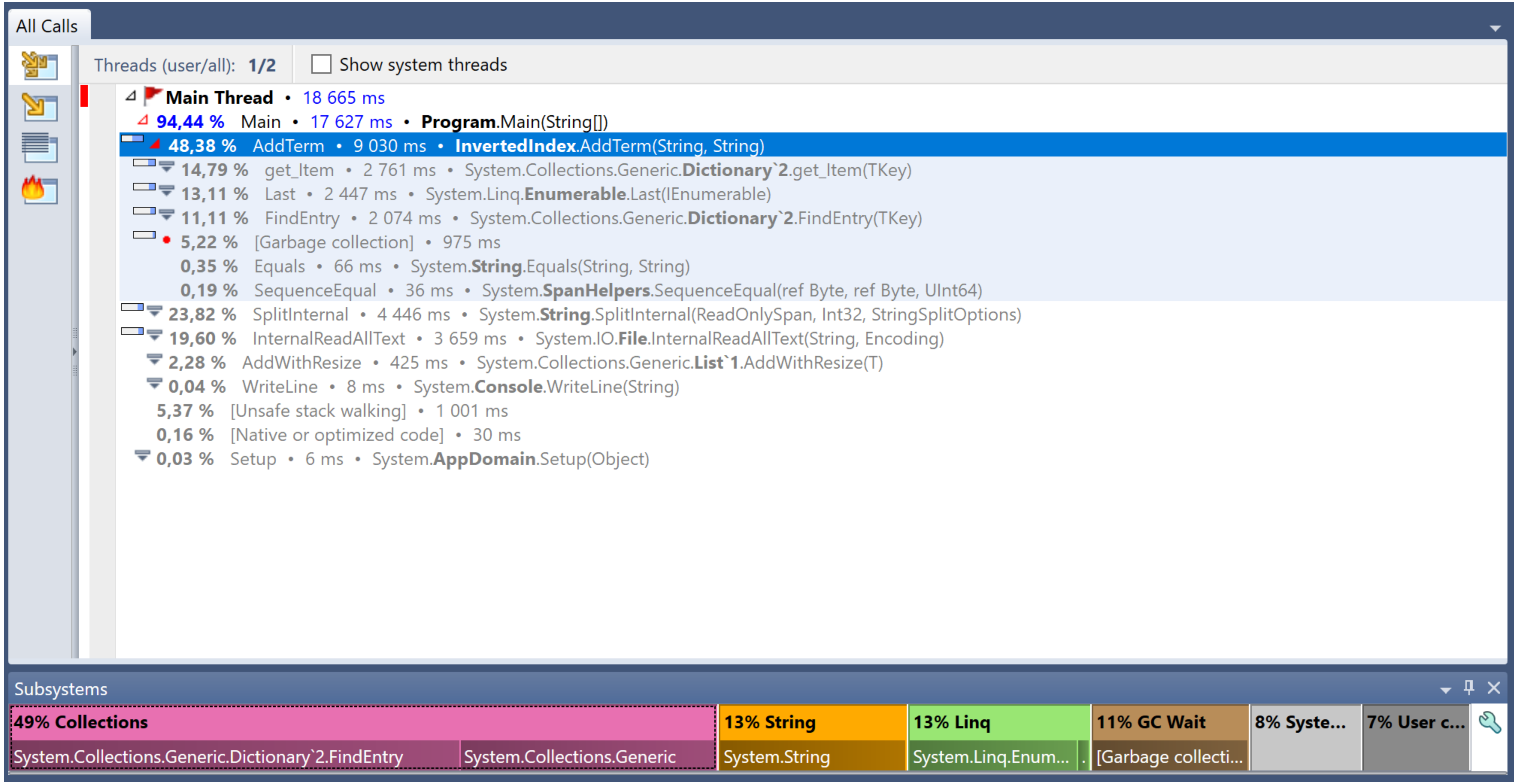
Tools

- Profiler
  - Observes a running program and reports performance metrics about it
  - Most can measure very granular: method based or line-by-line based
  - A lot of different metrics about CPU, memory usage
  - Most offer a UI to explore the results

- (Micro)Benchmarking library
  - Helps to test isolated parts of a program in an (as much as possible) controlled environment
  - Like a unit test, but for performance = easy to repeat
  - Runs a method multiple times and aggregates performance metrics

Profiler

- Intel VTune Amplifier Profiler https://software.intel.com/en-us/vtune
  - Allows for incredible detail, probably too much for most tasks
  - Supports quite a few languages: C, C++, C#, Fortran, Java, Python, Go, Assembly
- For C# (.NET): JetBrains dotTrace and dotMemory https://www.jetbrains.com/profiler/
- For every mainstream language in one form or another



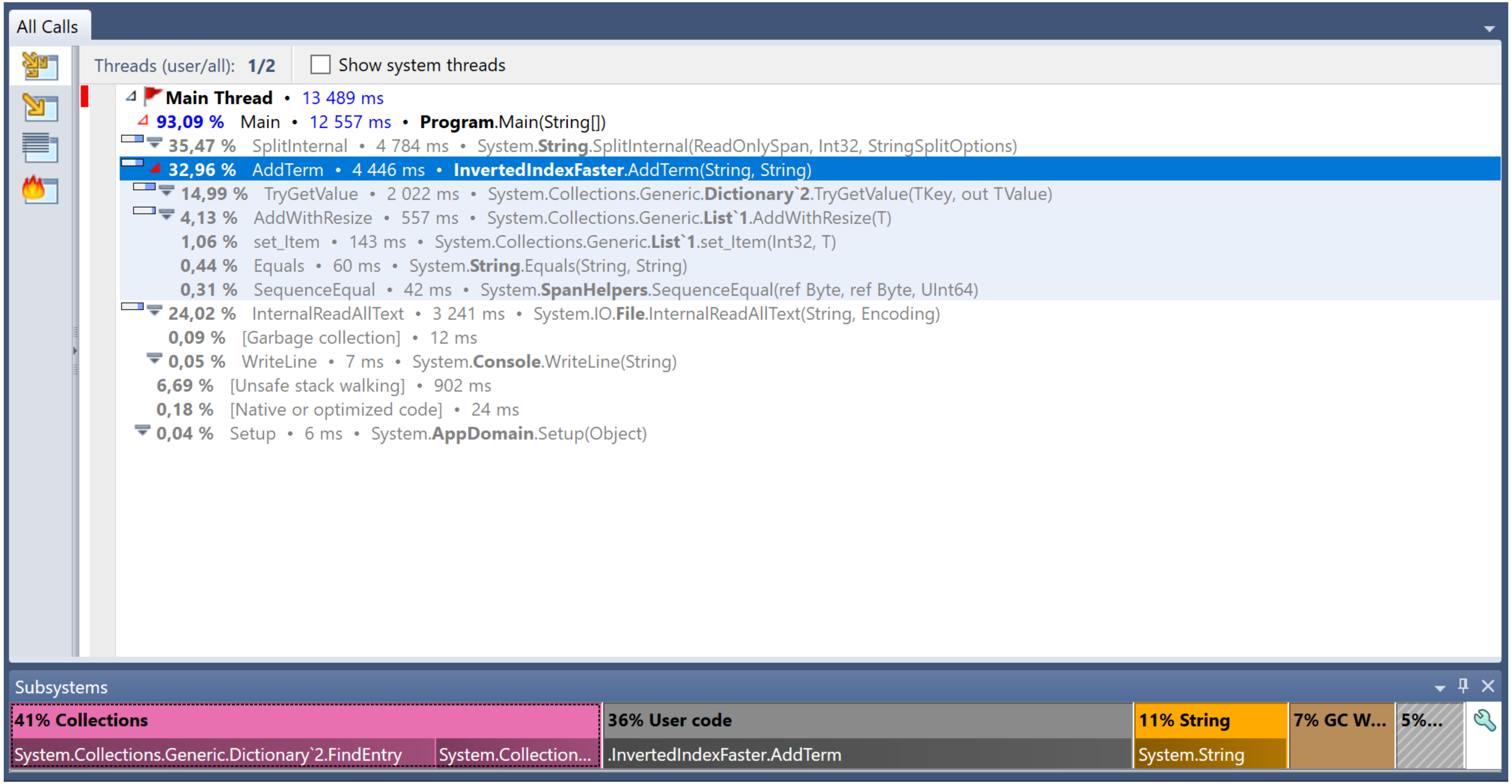
Threads (user/all): **1/2** ☐ Show system threads

```
◢ Main Thread • 18 665 ms
  ◢ 94,44 %  Main • 17 627 ms • Program.Main(String[])
    ◢ 48,38 %  AddTerm • 9 030 ms • InvertedIndex.AddTerm(String, String)
        14,79 %  get_Item • 2 761 ms • System.Collections.Generic.Dictionary`2.get_Item(TKey)
        13,11 %  Last • 2 447 ms • System.Linq.Enumerable.Last(IEnumerable)
        11,11 %  FindEntry • 2 074 ms • System.Collections.Generic.Dictionary`2.FindEntry(TKey)
        5,22 %  [Garbage collection] • 975 ms
        0,35 %  Equals • 66 ms • System.String.Equals(String, String)
        0,19 %  SequenceEqual • 36 ms • System.SpanHelpers.SequenceEqual(ref Byte, ref Byte, UInt64)
      23,82 %  SplitInternal • 4 446 ms • System.String.SplitInternal(ReadOnlySpan, Int32, StringSplitOptions)
      19,60 %  InternalReadAllText • 3 659 ms • System.IO.File.InternalReadAllText(String, Encoding)
      2,28 %  AddWithResize • 425 ms • System.Collections.Generic.List`1.AddWithResize(T)
      0,04 %  WriteLine • 8 ms • System.Console.WriteLine(String)
    5,37 %  [Unsafe stack walking] • 1 001 ms
    0,16 %  [Native or optimized code] • 30 ms
    0,03 %  Setup • 6 ms • System.AppDomain.Setup(Object)
```

49% Collections		13% String	13% Linq	11% GC Wait	8% Syste...	7% User c...
System.Collections.Generic.Dictionary`2.FindEntry	System.Collections.Generic	System.String	System.Linq.Enum...	[Garbage collecti...		

JetBrains dotTrace – call tree view – before optimization

https://github.com/sebastian-hofstaetter/teaching/tree/master/introduction-to-information-retrieval/inverted-index-example



Threads (user/all): **1/2** ☐ Show system threads

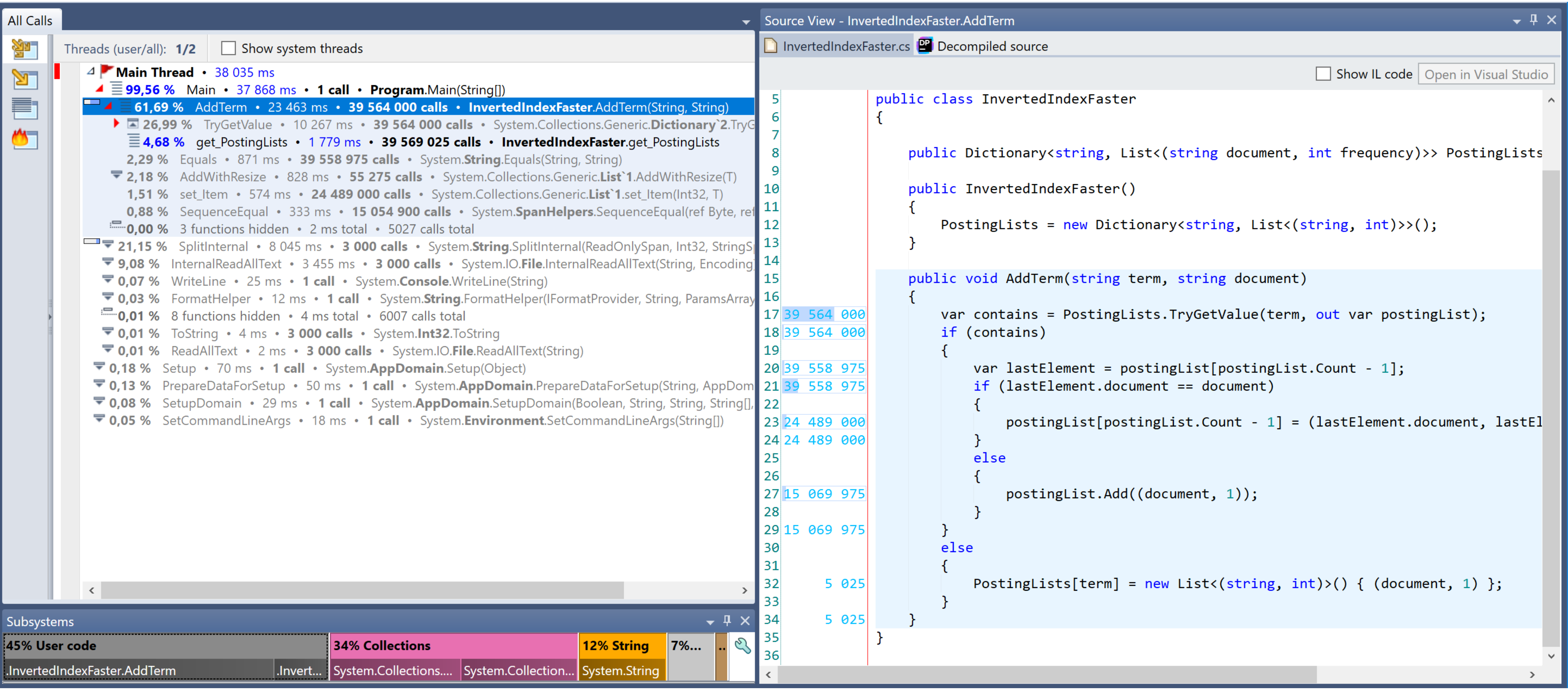
```
◢ Main Thread • 13 489 ms
 ◢ 93,09 %  Main • 12 557 ms • Program.Main(String[])
   35,47 %  SplitInternal • 4 784 ms • System.String.SplitInternal(ReadOnlySpan, Int32, StringSplitOptions)
   32,96 %  AddTerm • 4 446 ms • InvertedIndexFaster.AddTerm(String, String)
     14,99 %  TryGetValue • 2 022 ms • System.Collections.Generic.Dictionary`2.TryGetValue(TKey, out TValue)
     4,13 %  AddWithResize • 557 ms • System.Collections.Generic.List`1.AddWithResize(T)
     1,06 %  set_Item • 143 ms • System.Collections.Generic.List`1.set_Item(Int32, T)
     0,44 %  Equals • 60 ms • System.String.Equals(String, String)
     0,31 %  SequenceEqual • 42 ms • System.SpanHelpers.SequenceEqual(ref Byte, ref Byte, UInt64)
   24,02 %  InternalReadAllText • 3 241 ms • System.IO.File.InternalReadAllText(String, Encoding)
   0,09 %  [Garbage collection] • 12 ms
   0,05 %  WriteLine • 7 ms • System.Console.WriteLine(String)
  6,69 %  [Unsafe stack walking] • 902 ms
  0,18 %  [Native or optimized code] • 24 ms
  0,04 %  Setup • 6 ms • System.AppDomain.Setup(Object)
```

Subsystems

41% Collections		36% User code	11% String	7% GC W...	5%...
System.Collections.Generic.Dictionary`2.FindEntry	System.Collection...	.InvertedIndexFaster.AddTerm	System.String		

JetBrains dotTrace – call tree view – **after optimization**

https://github.com/sebastian-hofstaetter/teaching/tree/master/introduction-to-information-retrieval/inverted-index-example



Threads (user/all): 1/2 ☐ Show system threads

```
Main Thread • 38 035 ms
  99,56 %  Main • 37 868 ms • 1 call • Program.Main(String[])
    61,69 %  AddTerm • 23 463 ms • 39 564 000 calls • InvertedIndexFaster.AddTerm(String, String)
      26,99 %  TryGetValue • 10 267 ms • 39 564 000 calls • System.Collections.Generic.Dictionary`2.TryG
      4,68 %  get_PostingLists • 1 779 ms • 39 569 025 calls • InvertedIndexFaster.get_PostingLists
      2,29 %  Equals • 871 ms • 39 558 975 calls • System.String.Equals(String, String)
      2,18 %  AddWithResize • 828 ms • 55 275 calls • System.Collections.Generic.List`1.AddWithResize(T)
      1,51 %  set_Item • 574 ms • 24 489 000 calls • System.Collections.Generic.List`1.set_Item(Int32, T)
      0,88 %  SequenceEqual • 333 ms • 15 054 900 calls • System.SpanHelpers.SequenceEqual(ref Byte, ref
      0,00 %  3 functions hidden • 2 ms total • 5027 calls total
    21,15 %  SplitInternal • 8 045 ms • 3 000 calls • System.String.SplitInternal(ReadOnlySpan, Int32, StringSp
    9,08 %  InternalReadAllText • 3 455 ms • 3 000 calls • System.IO.File.InternalReadAllText(String, Encoding
    0,07 %  WriteLine • 25 ms • 1 call • System.Console.WriteLine(String)
    0,03 %  FormatHelper • 12 ms • 1 call • System.String.FormatHelper(IFormatProvider, String, ParamsArray
    0,01 %  8 functions hidden • 4 ms total • 6007 calls total
    0,01 %  ToString • 4 ms • 3 000 calls • System.Int32.ToString
    0,01 %  ReadAllText • 2 ms • 3 000 calls • System.IO.File.ReadAllText(String)
  0,18 %  Setup • 70 ms • 1 call • System.AppDomain.Setup(Object)
  0,13 %  PrepareDataForSetup • 50 ms • 1 call • System.AppDomain.PrepareDataForSetup(String, AppDom
  0,08 %  SetupDomain • 29 ms • 1 call • System.AppDomain.SetupDomain(Boolean, String, String, String[],
  0,05 %  SetCommandLineArgs • 18 ms • 1 call • System.Environment.SetCommandLineArgs(String[])
```

```
            public class InvertedIndexFaster
            {

                public Dictionary<string, List<(string document, int frequency)>> PostingLists

                public InvertedIndexFaster()
                {
                    PostingLists = new Dictionary<string, List<(string, int)>>();
                }

                public void AddTerm(string term, string document)
                {
39 564 000          var contains = PostingLists.TryGetValue(term, out var postingList);
39 564 000          if (contains)
                    {
39 558 975              var lastElement = postingList[postingList.Count - 1];
39 558 975              if (lastElement.document == document)
                        {
24 489 000                  postingList[postingList.Count - 1] = (lastElement.document, lastEl
24 489 000              }
                        else
                        {
15 069 975                  postingList.Add((document, 1));
                        }
15 069 975          }
                    else
                    {
     5 025              PostingLists[term] = new List<(string, int)>() { (document, 1) };
                    }
     5 025      }
            }
```

45% User code		34% Collections		12% String	7%...	..
InvertedIndexFaster.AddTerm	.Invert...	System.Collections....	System.Collection...	System.String		

JetBrains dotTrace – line-by-line analysis

Benchmarking library

```
public class Md5VsSha256 {
    private SHA256 sha256 = SHA256.Create();
    private MD5 md5 = MD5.Create();
    private byte[] data;

    [Params(1000, 10000)]
    public int N;

    [GlobalSetup]
    public void Setup() {
        data = new byte[N];
        new Random(42).NextBytes(data);
    }

    [Benchmark]
    public byte[] Sha256() => sha256.ComputeHash(data);

    [Benchmark]
    public byte[] Md5() => md5.ComputeHash(data);
}

public class Program {
    public static void Main(string[] args) {
        var summary = BenchmarkRunner.Run<Md5VsSha256>();
    }
}
```

- As an example: BenchmarkDotNet for C#
- Define a method to be benchmarked
  Runner takes care of the rest
- Comparable library for every major language available:
  C++: https://github.com/google/benchmark
  Java: http://openjdk.java.net/projects/code-tools/jmh
  …

Benchmarking tips

- Try not to compare apples and oranges
  - Use the same input, output & task

- Don’t run any other programs on your pc
  - Watch out for indexing, updating, etc... operations in the background

- If on a laptop: Plug it in + select the highest performance (Windows)

Garbage Collection

Memory management!

Memory locations: Stack & Heap

- Both are regions in memory

- Stack is small + fixed size
  - Used for local variables and other function related stuff
  - Calling a function puts the variables on the stack
    returning from a function removes them again
  - Fast, because all local offsets are known at compile time + add. optimizations

- Heap is used for allocating everything that does not fit into the stack
  - Either managed by “hand” = explicitly deleting objects
    or via an automatic garbage collector

Value & Reference types

- Value: data is located in the variable location (stack or inside and array)
- Reference: var contains pointer to actual data location (somewhere else)
  + many bytes overhead per instance (!)

- A lot of conceptual differences between languages
  - **C++** everything is put on the stack if you don't explicitly allocate
  - **Java, JavaScript, C#** primitive (value) types on the stack, all objects on the heap
  - **C#** allows to define new value types

- Also defines behavior of pass-by-value vs. pass-by-reference

Comparison: https://adamsitnik.com/Value-Types-vs-Reference-Types/

Garbage Collection 101

- Part of the runtime of memory managed languages (C#, Java, Python …)
- Keeps track of allocations and free memory on the heap

- Once an object is not used anymore (e.g. there is no reference pointing to it)
  - “garbage collects” it and sets the memory to be unused

- The GC is your friend!
- If your program spends too much time collecting garbage:

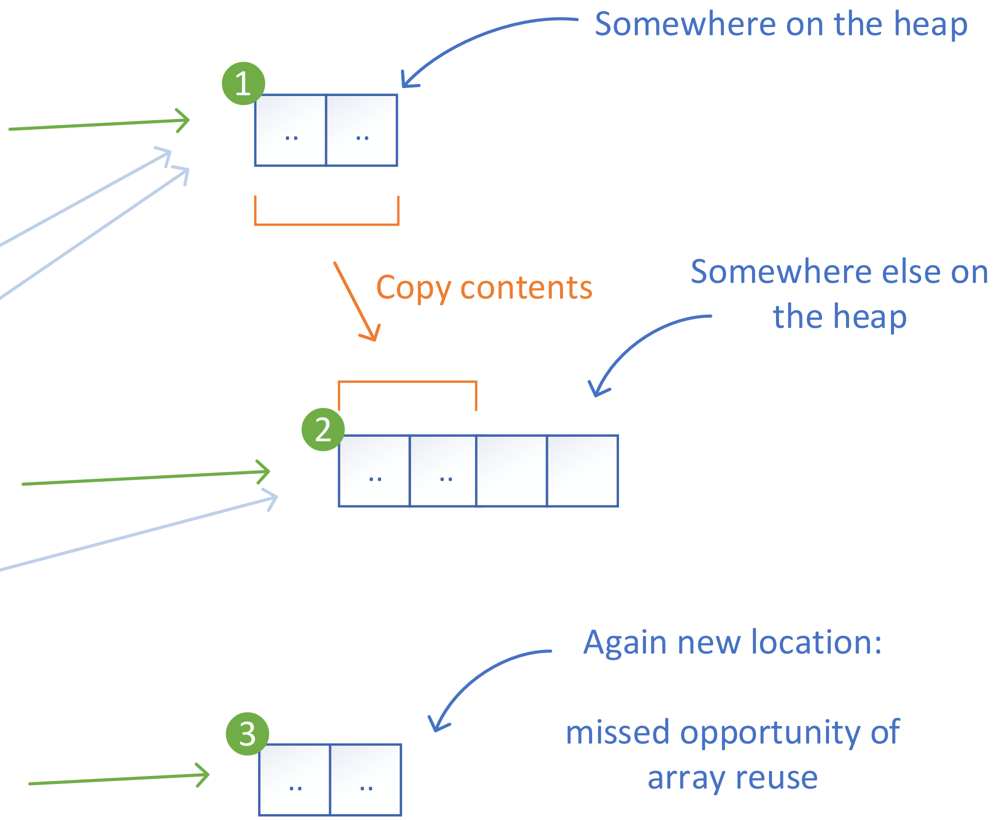
  Help the GC with domain specific knowledge and avoid unnecessary allocations

Garbage Collection 101: Example

- Implementing a posting list with: `List<T>` (self adjusting array based structure)

Indexing actions	`List<int>`
Term “cat”: Create instance:	Allocate new array Length: 0 Available: 2
1st doc:	Length: 1
2nd doc:	Length: 2
3rd doc:	Not enough space: allocate new array Length: 3 Available: 4
Garbage Collector	“delete” ①
Term “tiger”: Create instance:	Allocate new array

• • •



Somewhere on the heap

Copy contents

Somewhere else on the heap

Again new location:

missed opportunity of array reuse

Avoiding Garbage Collection

- Use an array pool
  - Easy to use: rent and return arrays
  - Hold a reference to an array until it is needed again
  - Good when you know you need arrays of same size multiple times

- Allocate one big block of memory and fill it up as you go along
  - For example: one big char array for your dictionary instead of many string objects
  - Handle an overflow before you overflow …

Research example: Realtime Posting Lists

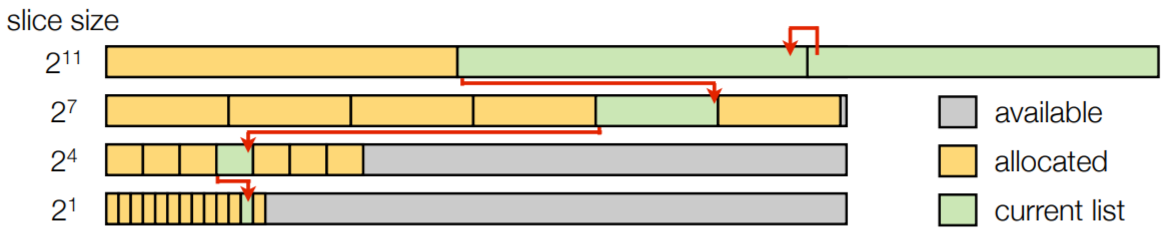


Fig. 2. Organization of the active index segment where tweets are ingested. Increasingly larger slices are allocated in the pools to hold postings. Except for slices in pool 1 (the bottom pool), the first 32 bits are used for storing the pointer that links the slices together. Pool 4 (the top pool) can hold multiple slices for a term. The green rectangles illustrate the the "current" postings list that is being written into.

- Realtime search needs
  - Updates available right away
  - High throughput
- Posting lists are allocated in a pool with different slice sizes
- Growing a posting list does not move the existing data

Busch et al. Earlybird: Real-Time Search at Twitter
http://users.umiacs.umd.edu/~jimmylin/publications/Busch_etal_ICDE2012.pdf

Strings are immutable! (C#, Java, and many others)

- Immutable: Can not change the contents of the data after initialization
- Operations on string objects result in copied memory + new string
  - Substring, split, …
- Copying objects results in more garbage collection
  - So you pay twice: once for copying/allocating and once for cleaning up

For more: https://lemire.me/blog/2017/07/07/are-your-strings-immutable/

The future: `Span<T>` (C# only for now)

- Abstracts away the memory region/type of the data
- Example: `Span<char>` unifies: `string`, `char[]` and `char*`
- Including language/framework/runtime integration
- Allows you to write fast, safe + memory efficient code

Introduction article: https://msdn.microsoft.com/en-us/magazine/mt814808.aspx

Performance improvements: https://blogs.msdn.microsoft.com/dotnet/2018/04/18/performance-improvements-in-net-core-2-1/

Memory mapped files

Let the operating system handle the memory

Memory mapped files

- A feature provided by the operating system
- Most languages support it (C++, Java, C#, Python …)

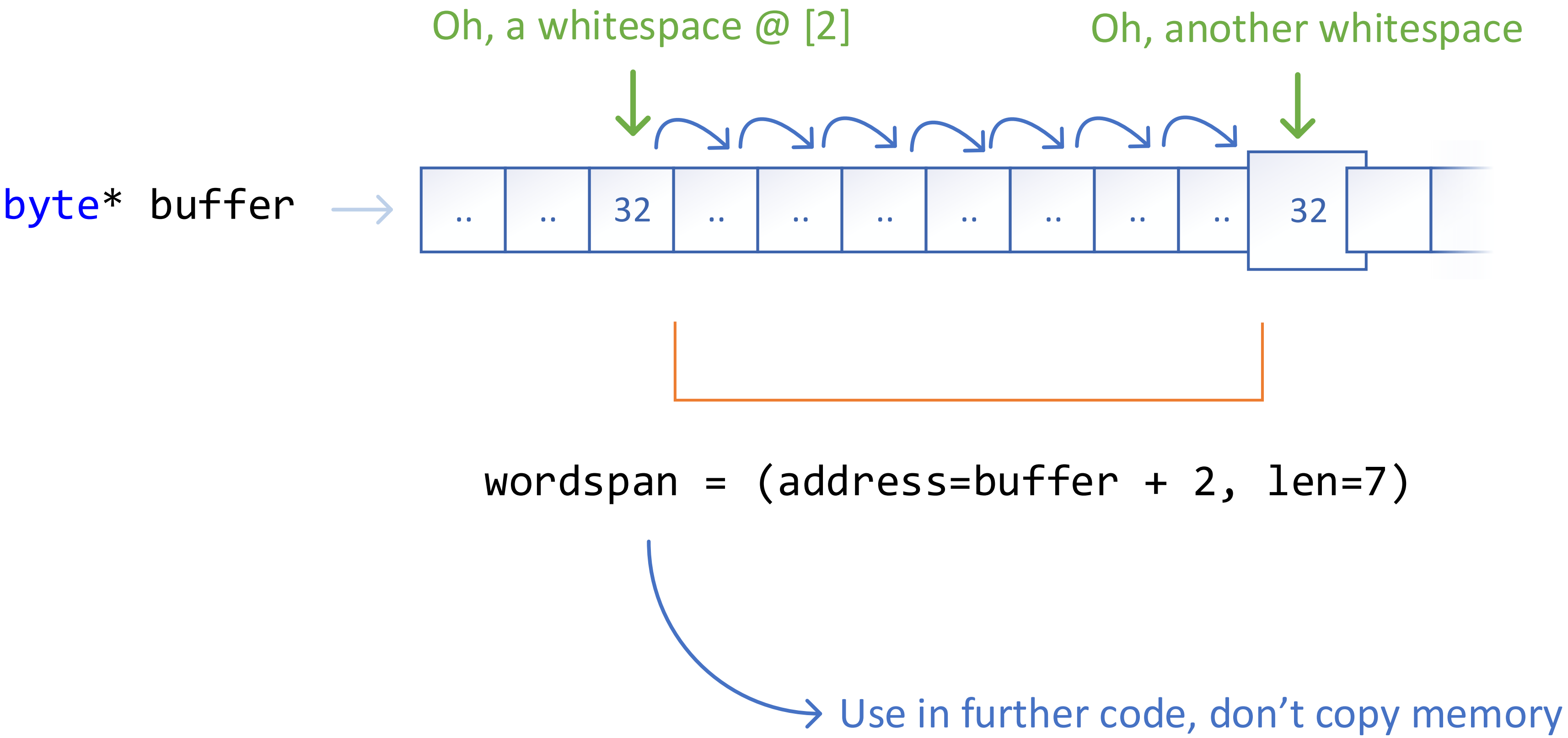
- OS puts the file contents in the RAM
- You get a pointer to this block of memory (~byte array)
  - You can sequentially go through your bytes (like a stream)
  - AND: also jump back a few bytes 😲 or completely randomly jump around
  - Read & write

Memory mapped files: C#

- It’s easy!

```
using (var mf = MemoryMappedFile.CreateFromFile(filepath, FileMode.Open))
using (var accessor = mf.CreateViewAccessor())
{
        byte* buffer = null;
        accessor.SafeMemoryMappedViewHandle.AcquirePointer(ref buffer);

        // use buffer as you want
        //  – including calling methods with it as parameter

        accessor.SafeMemoryMappedViewHandle.ReleasePointer();
}
```

Memory mapped files



Oh, a whitespace @ [2]

Oh, another whitespace

`byte* buffer`

```
wordspan = (address=buffer + 2, len=7)
```

Use in further code, don’t copy memory

Memory mapped files

- Beware: does not erase cost of I/O

- Makes additional buffering redundant when reading text
  - Removes pressure from the runtime GC (if used)
  - Makes it easier to view and process a file as byte[]

- Allows to read & write huge files
  - OS takes care of paging

- Also: Processes can share 1 memory mapped file

UTF-8 🌯

It’s not that easy …

UTF-8: The Format

- 1 – 4 bytes per character
- The 1st byte is compatible with ASCII + is uniquely identified as single char

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

- Byte[] length != how many characters you have

Working with raw UTF-8 bytes

- We don’t know all characters by looking at the raw bytes
    - But we can check them individually for ASCII characters (line breaks, whitespaces, …)
    - And search for beginning and end markers for an article in a file with many articles

- It is not necessary to transform a file of `byte[]` into a `string` (basically a `char[]`) as a whole

- Only decode parts you are interested in (e.g. tokens)

Convert byte array to useable string (C#)

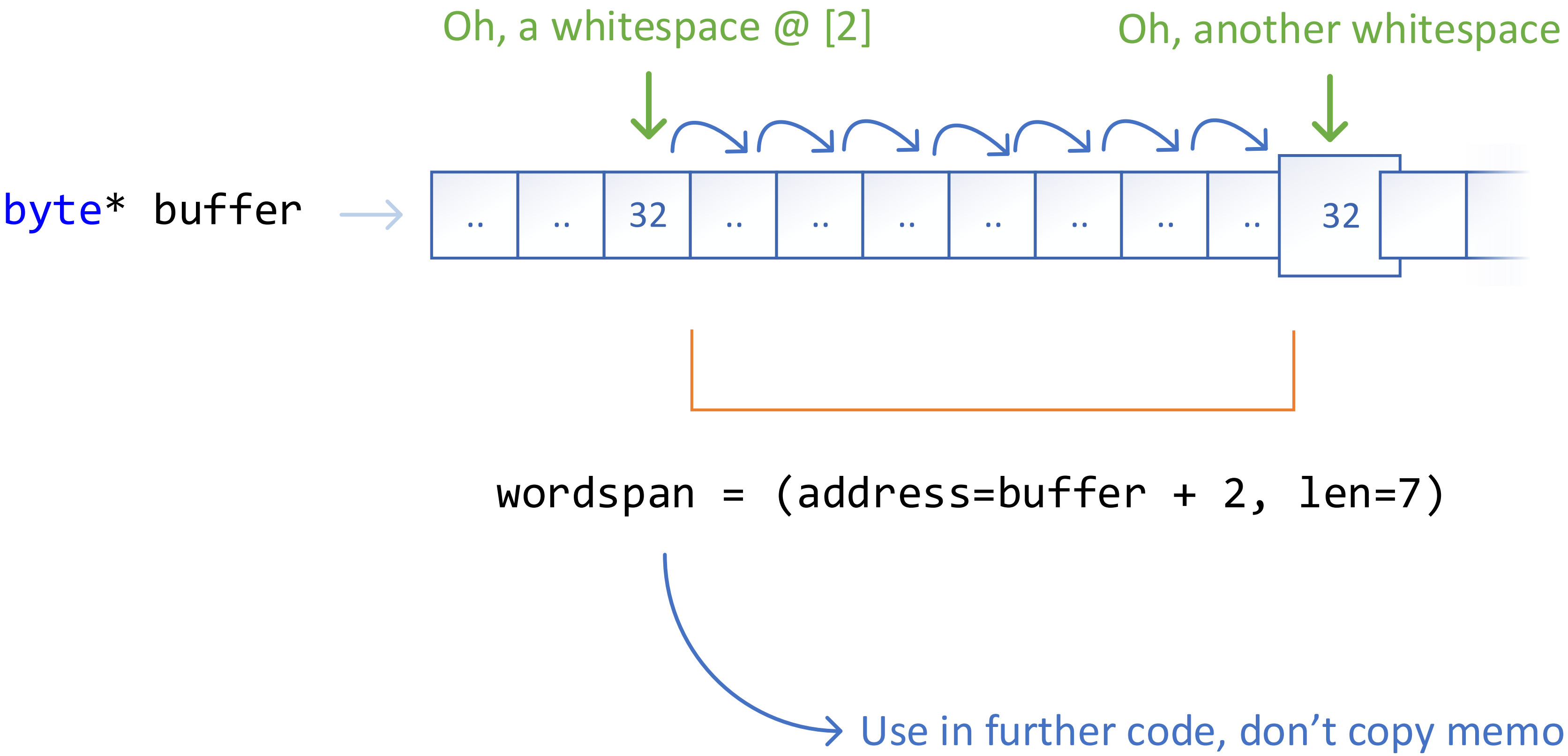
Allocating 2x duplicates !!

Allocating iterator

```
var text = Encoding.UTF8.GetString(buffer, doc.from, doc.until);
var words = text.Split(' ');

foreach (var word in words) {

        // process the word

}
```

Don’t do that (if you care about maximum performance)

Avoid creating objects



`wordspan = (address=buffer + 2, len=7)`

Use in further code, don’t copy memory

Avoid creating objects (C#)

```
char* wordBuffer = stackalloc char[100];
var currentWordStart = doc.from;

for (var i = doc.from; i < doc.until; i++) {

        if (*(buffer + i) == 32) { // split on whitespace character
                if (currentWordStart < i) {
                        var realLength = Encoding.UTF8.GetChars( bytes: buffer + currentWordStart,
                                                                 byteCount: i - currentWordStart,
                                                                 chars: wordBuffer,
                                                                 charCount: 100);
                        // process the word buffer with realLength

                }
                currentWordStart = i + 1;
        }
}
```

Using the right data structures

There are many solutions to a problem

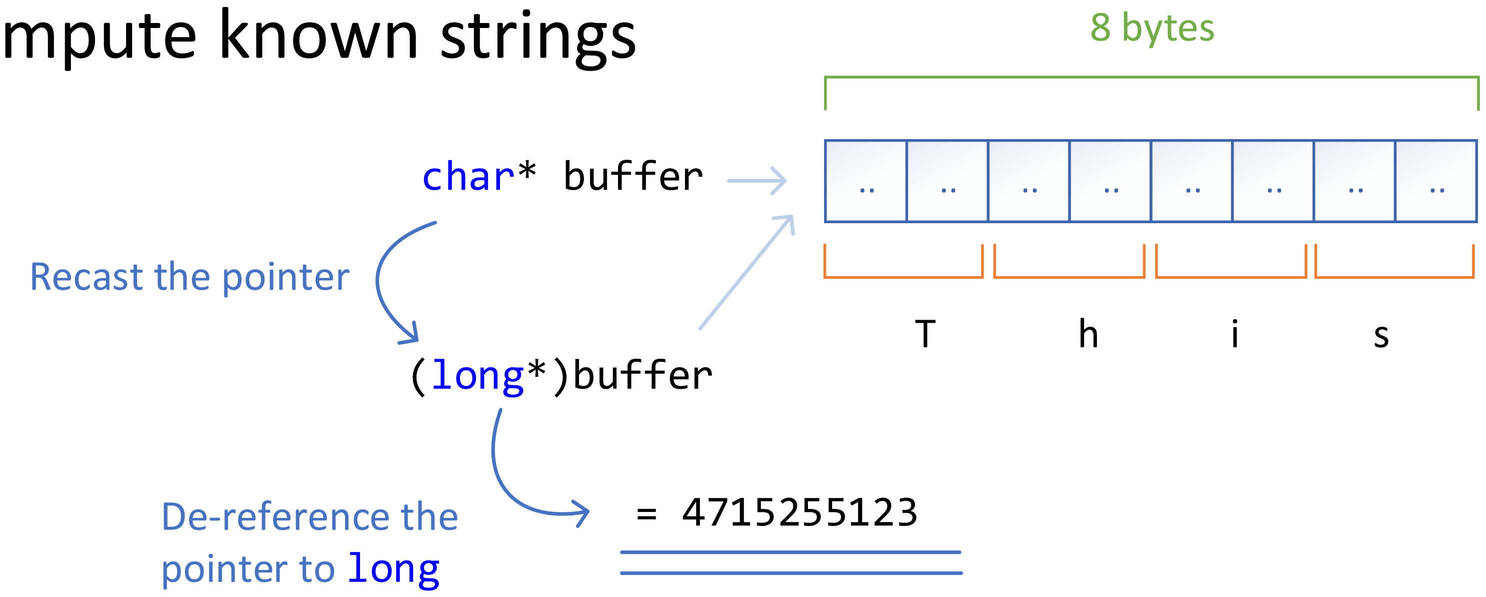
Strings vs. numbers

- Always use `int/long` when you can – Always !!
  - As an internal document id
  - As an internal word id (if you use it in more than the dictionary lookup)
  - In general: if you work with databases – never use a string key

- 64-bit system = 1 pointer to a string = 8 bytes = 1 long value
  - 1 String object takes up at least 8 bytes for the contents + 1 int (4 bytes) for the length + couple of other bytes for the object identity + function table …
  - And you save a random memory lookup

Matching known short strings

- A string with 4 chars = 8 byte = 1 long
- Just recast the pointer and interpret the 8 bytes as a long
- Pre-compute known strings


8 bytes

`char* buffer`

.. ..

T h i s

Recast the pointer

`(long*)buffer`

De-reference the pointer to `long`

`= 4715255123`

Idea from ASP.NET Core Web Server to get known http headers
https://github.com/aspnet/KestrelHttpServer/blob/release/2.2/src/Kestrel.Core/Internal/Infrastructure/HttpUtilities.cs

Finding stop words

- Most English stop words are <= 4 characters long
- Stop words: 10 to 100
- But we have to check them against every token (Wikipedia has 5 billion tokens)
- Can we do better than a `Set<string>`?
- Yes - Compare longs instead!

Benchmarks & Code @ https://github.com/sebastian-hofstaetter/search-engine/tree/master/benchmarks/stopwords

Finding stop words

Method	StopWordCount	Mean	Error	StdDev	Scaled
StringSet	50	34.277 ns	0.2420 ns	0.2263 ns	1.00
LongIteration	50	20.633 ns	0.0705 ns	0.0589 ns	0.60
LongHashSet	50	13.380 ns	0.0391 ns	0.0327 ns	0.39
LongBinarySearch	50	22.290 ns	0.0495 ns	0.0439 ns	0.65
LongTreeSet	50	21.182 ns	0.0599 ns	0.0531 ns	0.62
StopWordSet	50	**3.081 ns**	0.0377 ns	0.0353 ns	**0.09**

Results from https://github.com/sebastian-hofstaetter/search-engine/tree/master/benchmarks/stopwords

Parallelization

e.g. Multithreading

Multithreading tips

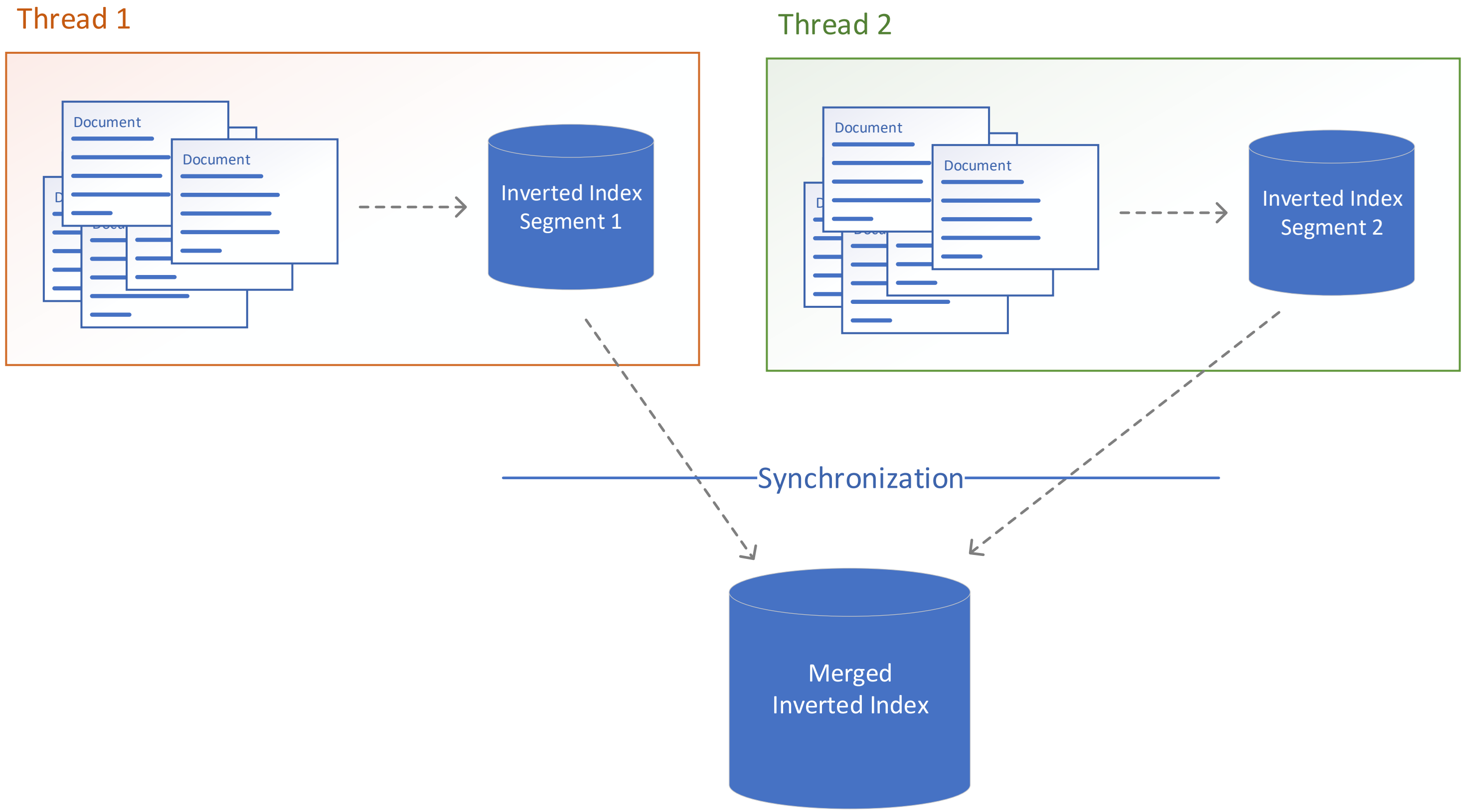
- In general multithreading improves indexing time
  - Especially with SSD disks (I/O becomes less of a bottleneck)

- Locking, context switches, synchronization, etc… all take time

- Try to do as much independent work per thread as possible
  - For example: Let 1 thread work on 1 index and 1 document independent from the other threads

Multithreaded index creation example

- This is one possibility, there are countless others you might try



Atomic operations

- Problem: x = x + 1 is not thread safe (!)
  - Is actually: load x in y, add y + 1, move y into x
  - This can lead to a “+ 1” just jumped over by another thread

- Atomics: combine load, transform, move into single instruction
  - Making the operation thread safe
  - Supports easy add, increment, exchange operations

See: C++ https://software.intel.com/en-us/node/506090
C# https://docs.microsoft.com/en-us/dotnet/standard/threading/interlocked-operations

Also supported by Java and many other languages

Summary: Efficient & Fast Text Processing

1. No matter what you do measure it
2. Try to avoid copying large amounts of memory
3. Read some articles about performance optimization for your stack

1. No matter what you do measure it
2. Try to avoid copying large amounts of memory
3. Read some articles about performance optimization for your stack

Thank You