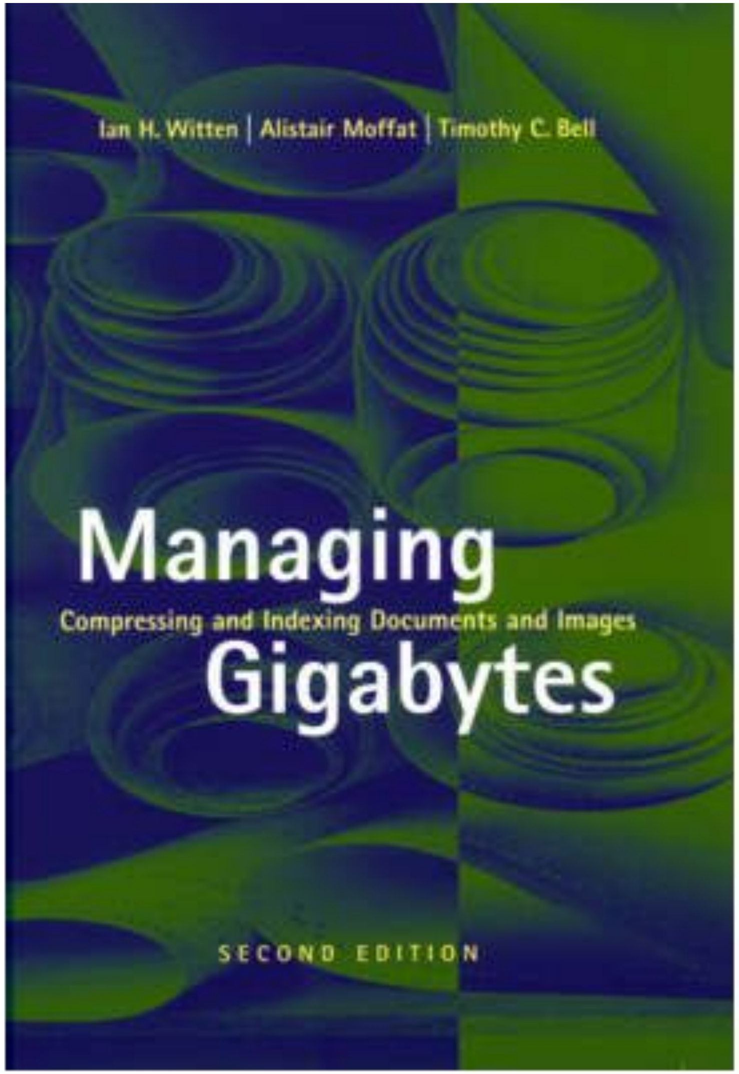
University of Helsinki
Department of Computer Science

582487
Data Compression Techniques

Lecture 4: Integer Codes II

Simon J. Puglisi
(puglisi@cs.helsinki.fi)

Books (again)…



Ian H. Witten | Alistair Moffat | Timothy C. Bell

Managing Gigabytes

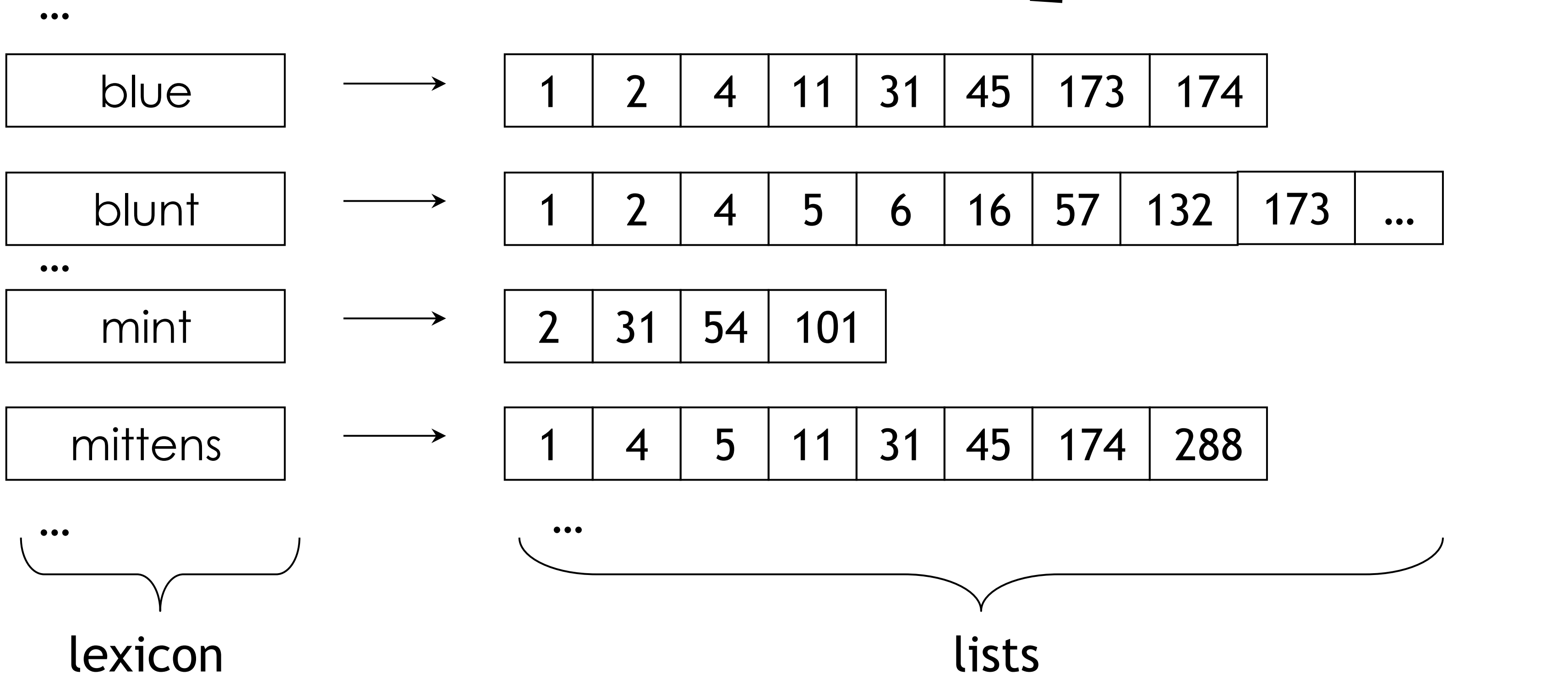
Compressing and Indexing Documents and Images

SECOND EDITION

Remember how Google works?

- Crawl the web, gather a collection of documents
- For each word t in the collection, store a list of all documents containing t:
- Query: blue mittens

Inverted Index



...

blue → 1 | 2 | 4 | 11 | 31 | 45 | 173 | 174

blunt → 1 | 2 | 4 | 5 | 6 | 16 | 57 | 132 | 173 | ...

...

mint → 2 | 31 | 54 | 101

mittens → 1 | 4 | 5 | 11 | 31 | 45 | 174 | 288

...

...

lexicon

lists

- The key idea for compressing inverted lists is to observe that elements of *L* are monotonically increasing, so we can transform the list by taking differences (gaps):

  L = 3,7,11,23,29,37,41,…

  D(L) = 3,4,4,12,6,8,4,…

- The advantage of doing this is to make the integers smaller, and suitable for integer codes
  - Remember we said smaller integers get smaller codes

- We can easily “degap” the list during intersection
  - Adding items together as we scan the lists adds little overhead

Outline

- Tuesday: three classic integer codes
  - Unary
  - Elias codes (gamma, delta)
  - Golomb codes (Rice, general)

- Today: three modern flavours
  - Interpolative binary codes
  - Variable-byte codes
  - Word-aligned binary codes (simple, relative, carryover)

Interpolative codes…

Interpolative codes

- Invented by Moffat and Stuiver, 2000
- Suitable for a list of increasing integers
  - Like the lists in an inverted index we were discussing last time
- Treats the list of integers to be encoded holistically
- Capable of exploiting clustering present in the input
- Input: array of integers
- Output: a sequence of bits

Interpolative codes

- Consider the following list of integers

  L = 2,9,12,14,19,21,31,32,33

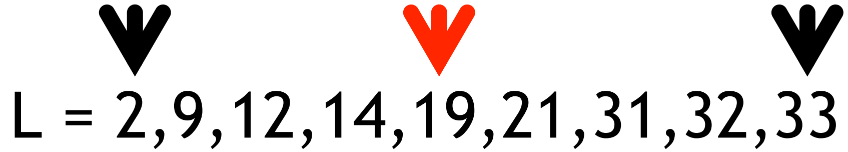
- Interpolative coding compresses this list by encoding its
  - length (9), and
  - its first element L[1] = 2, and
  - Its last elements and L[9] = 33,
  - using some other method, say γ coding.

    $\gamma(9)$, $\gamma(2)$, $\gamma(33)$

- It then proceeds to encode the middle item: L[5] = 19

- However, when encoding L[5] it exploits that L[1] and L[9], and the length are already known to the decoder

Interpolative codes



L = 2,9,12,14,19,21,31,32,33

- Our integers are stored in increasing order:

  2 = L[1] < L[2] < … < L[5] < … L[8] < L[9] = 33.

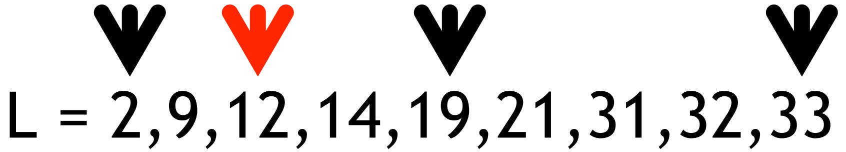
- Therefore, based on the the values of L[1] and L[9], L[5] must lie in the interval [2,33]

- In fact, because there are 9 elements in the list we can say something stronger about L[5]… it is in [6,29].

- This interval contains 24 distinct values, of which L[5] is the 13th (counting from 0).

- So we encode L[5]=13 using $2^{ceil(log(24))}$ = 5 bits as 01101.

Interpolative codes



L = 2,9,12,14,19,21,31,32,33

- We then proceed recursively… to encode L[3]

- L[3] can be coded with the knowledge that the decoder now knows about L[5], which forces L[3]'s value to be in the range [4,17]

- It is the 8th of those 14 numbers, so we encode 8 in $2^{\text{ceil}(\log(14))}$ = 4 bits.

Interpolative codes

Original order	Encoding order	Bits	Description
n=9	n=9	1110001	$\gamma(9)$
2	2	100	$\gamma(2)$
9	33	111101111	$\gamma(33)$
12	19	01101	13=19-6 in 5 bits
14	12	1000	8=12-4 in 4 bits
19	9	0110	6=9-3 in 4 bits
21	14	001	1=14-13 in 3 bits
31	31	1010	10=31-21 in 4 bits
32	21	0001	1=21-20 in 4 bits
33	32	★L[8] = 32 is encoded using 0 bits	

Nice things about Interpolative codes

- Interpolative coding effectively exploits clustering present in the integers being encoded

- To encode f integers all in the range [1,N], in the worst case interpolative coding uses

  $$f \bullet (2.58 + \log (N/f)) \text{ bits.}$$

- This compares nicely with the best case for Golomb coding: $f \bullet (1.44 + \log (N/f))$ bits

- … and interpolative coding is parameter free.

Digression: clustering in Search Engines

- There are (at least) two ways in which document ids can become clustered inside the term lists
- Firstly, we can try to *induce it* before building the index by, for example, ordering the documents by URL
  - Within a domain (say helsinki.fi, ford.com) documents use a similar vocabulary
- Secondly, it can be *natural*.
  - Think of trending terms on Twitter.

Moving away from bits…

- The methods we have discussed so far all involve examining ***lots*** of individual bits
  - esp. the Elias and Golomb codes from last time, both of which involve unary codes

- Such “bit fiddling” limits the rate of decoding

- Next we look at two methods that are specifically designed with high decoding throughput in mind

Variable byte codes…

Variable byte (VB) code

- Developed by Scholer et al., 2002
- Used by many commercial/research systems
- Good blend of variable-length coding and sensitivity to memory alignment (bit-level codes, see last week).
- Input: array of integers
- Output: array of bytes

Variable byte (VB) code

- Dedicate 1 bit in each byte we output (high bit) to be a continuation bit c.
  - 00000000

- If the int G fits within 7 bits, binary-encode it in the 7 available bits and set c = 0.
  - Eg. Integer 29 = 11101 fits in 7 bits so we output: 00011101
  - Eg. integer 117 = 1110101 fits in 7 bits so we output: 01110101

- Else: set c = 1, encode lower-order 7 bits and then use additional bytes to encode the higher order bits using same algorithm.
  - Eg. integer 767 = 1011100101 > 7 bits so:
  - Put lower 7 bits (1100101) in first byte and set the c bit: 11100101
  - We now have the bits 101 to deal with, < 7 bits so output 00000101

- At the end, the continuation bit of the last byte is 0 (c = 0) and the other bytes is 1 (c = 1). We know when we have decoded an item!

Variable byte (VB) code

- Another example:
- 214577
- 110100011000110001
- 0110001 → 10110001
- 0001100 → 10001100
- 1101 → 00001101

- 10110001
- 10001100
- 00001101
- 000110100011000110001 = 214577

VB code examples

docIDs	824	829	215406
gaps	824	5	214577
in binary	1100111000	101	1101000110 00110001
VB codes	10111000 00000110	00000101	10110001 10001101 00001100

VB encoded list:

101110000000011000000101101100011000110100001100

101110000000011000000101101100011000110100001100

VB code encoding algorithm

```
VBEncodeNumber(n)
    bytes ← {}, i ← 0
    while n >= 128 do
        bytes[i] ← 128 + (n mod 128)
        n ← n div 128
        i ← i + 1
    end while
    bytes[i] ← n
    return bytes
```

VB code decoding algorithm

```
VBDecodeNumber(bytes)
    $n \leftarrow 0, i \leftarrow 0$
    $d \leftarrow 1$
    while $bytes[i] \geq 128$ do
        $n \leftarrow n + d * (bytes[i] - 128)$
        $d \leftarrow d * 128$
        $i \leftarrow i + 1$
    end while
    $n \leftarrow n + d * bytes[i]$
    return $n$
```

VB code decoding algorithm (with bit shifting)

```
VBDecodeNumber(bytes)
    $n \leftarrow 0, i \leftarrow 0$
    $d \leftarrow 1$
    while $bytes[i] \geq 128$ do
        $n \leftarrow n + ((bytes[i] \& 127) << shift)$
        $shift \leftarrow shift + 7$
        $i \leftarrow i + 1$
    end while
    $n \leftarrow n + (bytes[i] << shift)$
    return $n$
```

Other variable length codes

- Instead of bytes, we could use a smaller “unit of alignment”: e.g., 4 bits (nibbles), two per byte.

- Variable byte alignment wastes space if you have many small gaps – nibbles do better on those.

- Of course we can also try to make use of larger units of alignment, such as words…

Word-aligned binary codes…

Information Retrieval, 8, 151–166, 2005

Inverted Index Compression Using Word-Aligned Binary Codes

VO NGOC ANH
ALISTAIR MOFFAT alistair@cs.mu.oz.au
Department of Computer Science and Software Engineering, The University of Melbourne, Victoria 3010, Australia

Received June 18, 2003; Revised December 24, 2003; Accepted January 6, 2004

Abstract. We examine index representation techniques for document-based inverted files, and present a mechanism for compressing them using word-aligned binary codes. The new approach allows extremely fast decoding of inverted lists during query processing, while providing compression rates better than other high-throughput representations. Results are given for several large text collections in support of these claims, both for compression effectiveness and query efficiency.

Keywords: index compression, integer coding, index representation

Word-aligned binary code

- Invented by Ahn & Moffat in 2005
- Very vague idea is to have a hybrid between bit-aligned and byte-aligned codes
- Allows code words as short as one bit, but forces a regular pattern within each compressed word
  - This greatly reduces decoding costs compared to bit-level codes
- Sensitive to local clustering present in the input stream
  - (As interpolative coding is)
- Input: array of integers; output: array of 32-bit words

Simple-9: our first word-aligned binary code

- Philosophy: try to assign the maximum possible of number of integers from the input to each output word.

- Each 32-bit word stores *a number of* integers

- Different words store different numbers of integers, but within each word each integer is represented using exactly the same number of bits.

Simple-9: our first word-aligned binary code

- In the Simple-9 scheme, 28 out of the 32 bits in a word are reserved for holding integers.

00000000000000000000000000000000

- e.g., if the next 28 integers in the list are all 1 or 2, then 28 codes of 1-bit each can be used to code them
  - (recall we're assuming the integers are all strictly > 1)
- At the other extreme, if the next integer is > 2^{14}, then it must be coded into a single word as a 28-bit binary code
  - No room in 28 bits for more than one integer > 2^{14}.
  - Between these extremes, seven 4-bit codes, four 7-bits, et c.
- Remaining four selector bits tell us how the 28 "data bits" are partitioned

Nine different ways of using 28 bits for flat binary codes (method Simple-9)

Selector	Number of codes	Length of each code (bits)	Number of unused bits
a	28	1	0
b	14	2	0
c	9	3	1
d	7	4	0
e	5	5	3
f	4	7	0
g	3	9	1
h	2	14	0
i	1	28	0

Simple-9 in action (1)

- To see this process in action, consider the following ints

  (4,6,1,1,3,5,1,7,1,13,20,1,12,20)

- If applicable, row ***a*** yields the most compact representation, but because there is a value > 2 = 2^1 in the first 28 ints to be coded, row ***a*** cannot be used. (For similar reasons we can't use row ***b*** either)

Simple-9 in action (2)

- To see this process in action, consider the following ints

  (4,6,1,1,3,5,1,7,1,13,20,1,12,20)

- But none of the first 9 entries are greater than 8 = 2^3, so we can use row ***c*** to drive the first word of output

Simple-9 in action (3)

- To see this process in action, consider the following ints

  (4,6,1,1,3,5,1,7,1,13,20,1,12,20)

- But none of the first 9 entries are greater than 8 = 2^3, so we can use row ***c*** to drive the first word of output

- Each of the first 9 entries go into the word with 3-bit codes

  c, 011, 101, 000, 000, 010, 100, 000, 110, 000, .,

  (an integer of 1 maps to a code of 000)

Simple-9 in action (4)

- To see this process in action, consider the following ints

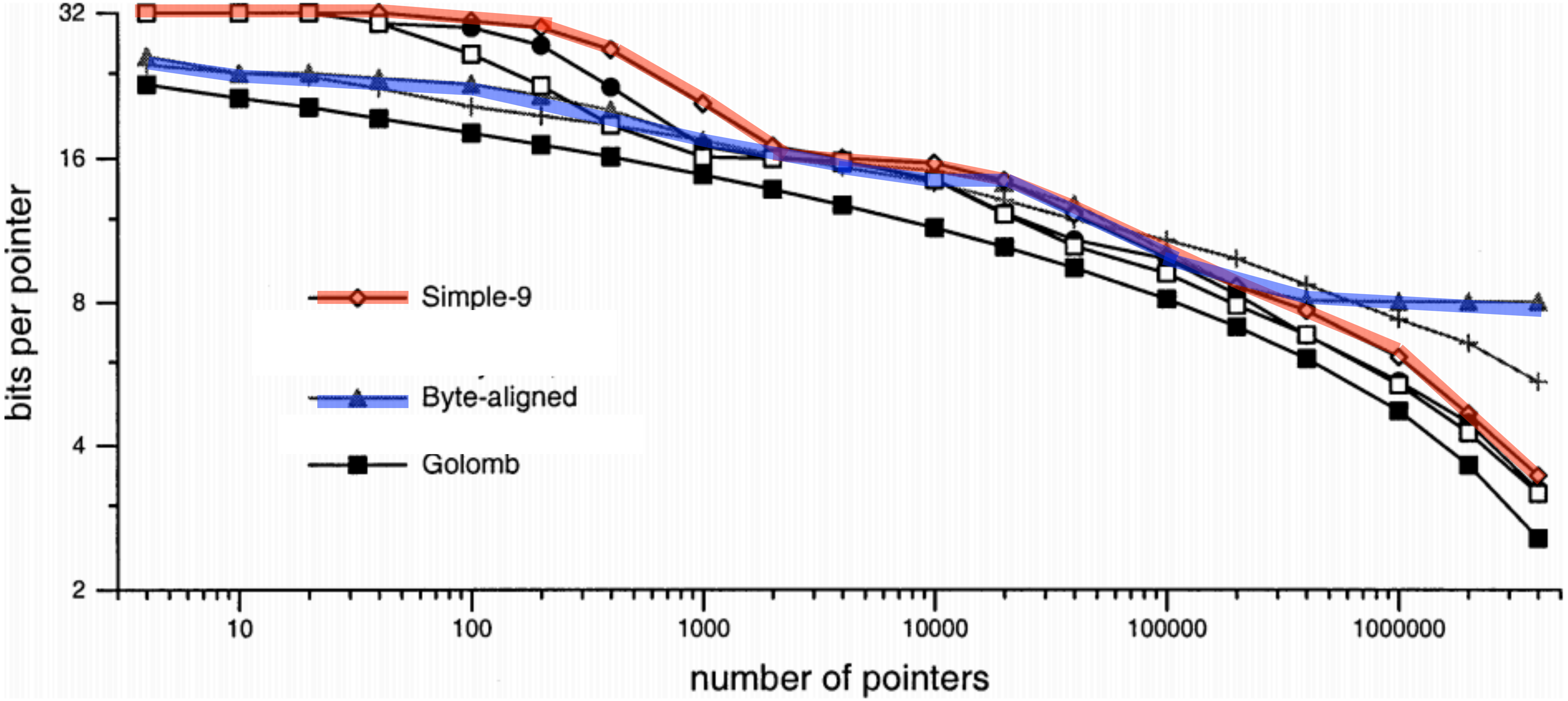
  (4,6,1,1,3,5,1,7,1,13,20,1,12,20)

- The remaining values (starting at 13, with five further values to be coded) yields a second output word:

  e, 01100, 10011, 00000, 01011, 10011, ...,

- where now three bits are unused in the 32-bit code
- Overall: 64 bits for 14 ints. A Golomb code would use 58.

Simple-9: Compression Performance



Addressing Simple-9's shortcomings

- Simple-9's use of 4 bits for the selector means 7 of 16 possible selector combinations are unused – a loss of almost 1 bit per word

- One way to recover this loss would be to eliminate a row in the table (row ***e***?) and choose among 8 possibilities for each word with a 3 bit selection code.
  - Would allow 29 bits of data in each word… but 29 is prime ☹

- We could introduce a variable length selector…
  - 2 bit selector => 2 x 15-bit codes
  - 3 bit selector => 9 x 3-bit codes, et c.

- …but this reintroduces bit fiddling.

Relative-10: improving on Simple-9

- A more interesting approach is the shrink the selector to just 2 bits, leaving 30 data bits.
  - 30 has many factors -> a wider range of partitionings than 28

Different ways of using 30 bits for flat binary codes (method Relative-10)

Selector	Number of codes	Length of each code (bits)	Number of unused bits
a	30	1	0
b	15	2	0
c	10	3	0
d	7	4	2
e	6	5	0
f	5	6	0
g	4	7	2
h	3	10	0
i	2	15	0
j	1	30	0

Relative-10: improving on Simple-9

- A more interesting approach is the shrink the selector to just 2 bits, leaving 30 data bits.
  - 30 has many factors -> a wider range of partitionings than 28
- The problem now is the selector.
  - With only 2 bits we are restricted to just 4 choices for each word
- To make the most of those 4 combinations, they are interpreted *relative* to the selector of the previous word: *one row less*; *same row*; *one row more*; *last row*

Relative-10: improving on Simple-9

- i.e. if the previous row make use of row *r*, then the 2 bit selector of the current word indicates on of row r-1, r, r+1, and row j.
  - At the extremities of the table (r=a and r=j) choices are altered to always provide 4 viable alternatives

- Encoder and decoder must agree on an initial value of the “current selector”

Transfer matrix for selectors in Relative-10

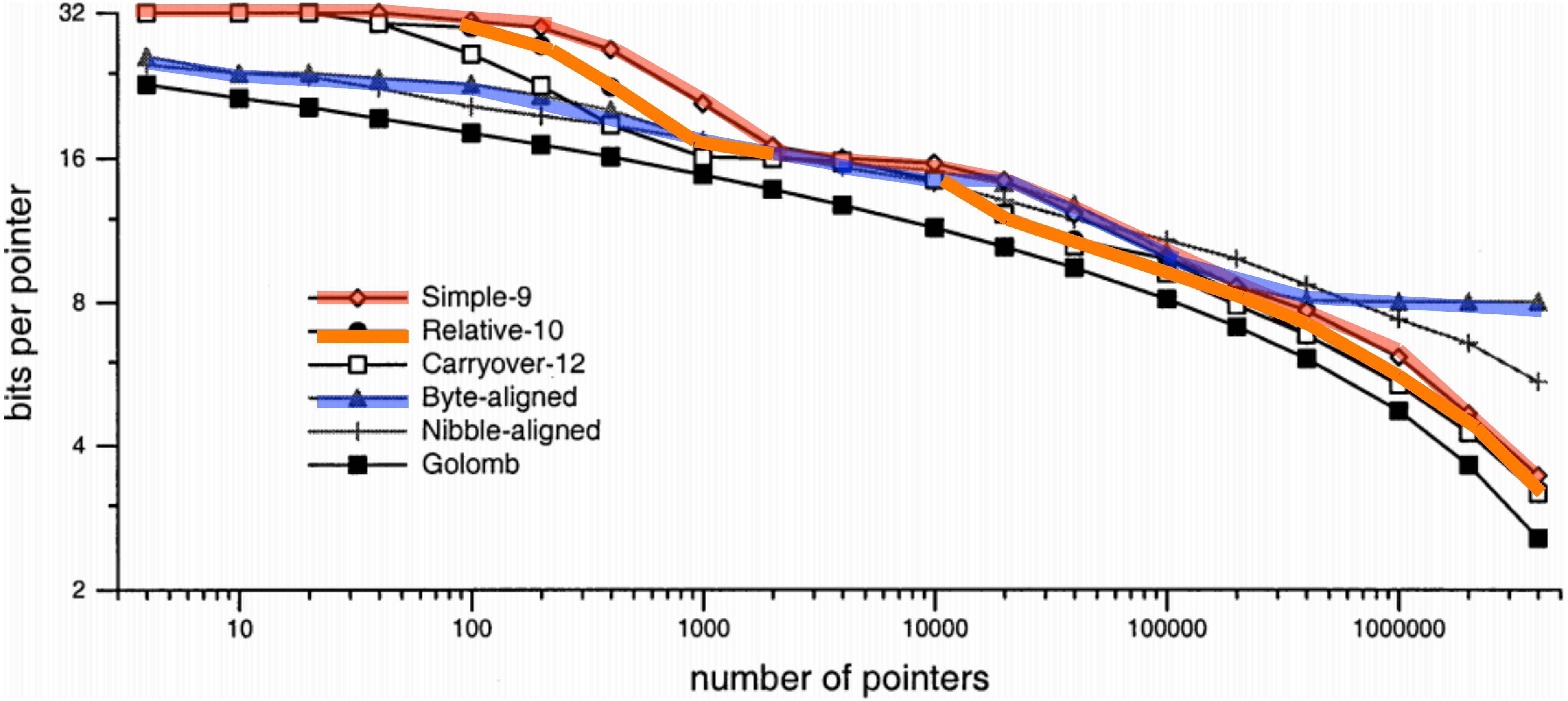
	Possible next selector values									
Current Selector	a	b	c	d	e	f	g	h	i	j
a	0	1	2							3
b	0	1	2							3
c		0	1	2						3
d			0	1	2					3
e				0	1	2				3
f					0	1	2			3
g						0	1	2		3
h							0	1	2	3
i							0	1	2	3
j							0	1	2	3

Relative-10: improving on Simple-9

- If the set of integers being coded is homogeneous, the bulk of integers will get coded using an appropriate row

- The occasional large int may result in a sudden shift to row j, and the consequent use of some long code words, but the current state can then migrate back up the table to the natural position for the list

Relative-10: Compression Performance



A final squeeze and then goodbye

- Look again at the table of selectors used by Relative-10

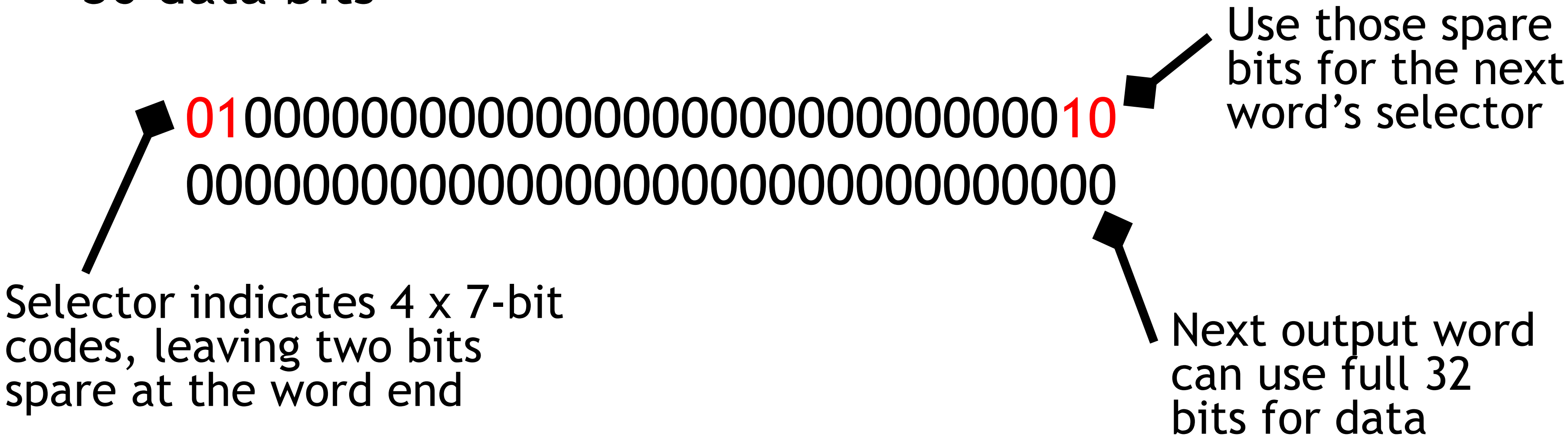
Selector table for Relative-10 (again)

Selector	Number of codes	Length of each code (bits)	Number of unused bits
a	30	1	0
b	15	2	0
c	10	3	0
d	7	4	2
e	6	5	0
f	5	6	0
g	4	7	2
h	3	10	0
i	2	15	0
j	1	30	0

Carryover-12: A final squeeze and then goodbye

- When 7-bit and when 4-bit codewords are in use, there are two spare bits in each word.

- Our next method – *Carryover-12* – uses those spare bits to store the selector value for the next word of codes, thereby allowing that next word to use 32 rather than 30 data bits



```
01000000000000000000000000000010
00000000000000000000000000000000
```

Use those spare bits for the next word’s selector

Selector indicates 4 x 7-bit codes, leaving two bits spare at the word end

Next output word can use full 32 bits for data

Carryover-12: two more tweaks

- Also, because the selector in relative-10 is relative, we are free to add rows to the table, to improve the precision of the code words assigned…

Selector table for Relative-10 (again)

Selector	# of codes	Code length	Unused bits
a	30	1	0
b	15	2	0
c	10	3	0
d	7	4	2
e	6	5	0
f	5	6	0
g	4	7	2
h	3	10	0
i	2	15	0
j	1	30	0

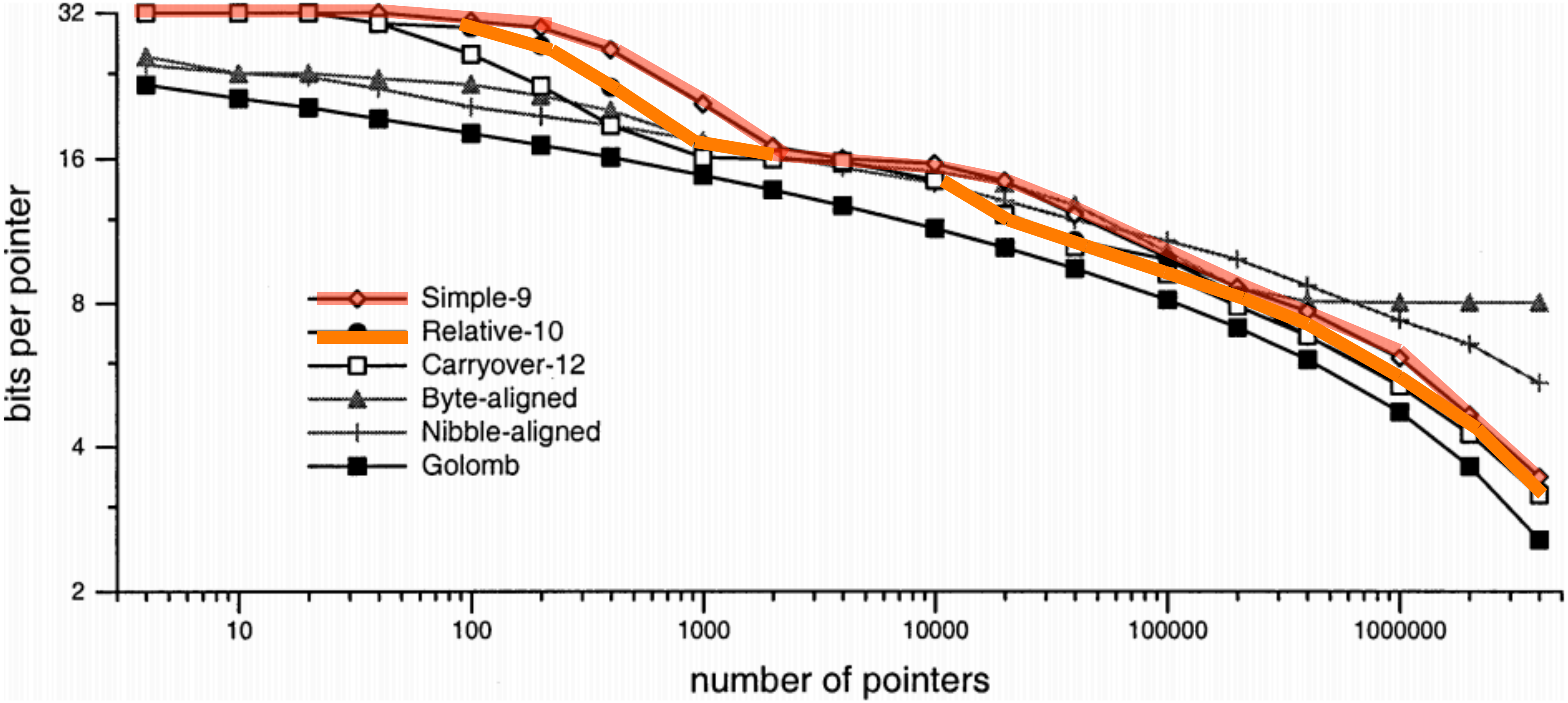
Selector table for Relative-10 (again)

Selector	# of codes	Code length	Unused bits
a	30	1	0
b	15	2	0
c	10	3	0
d	7	4	2
e	6	5	0
f	5	6	0
g	4	7	2
h	3	10	0
i	2	14	2
j	2	15	0
k	1	30	0

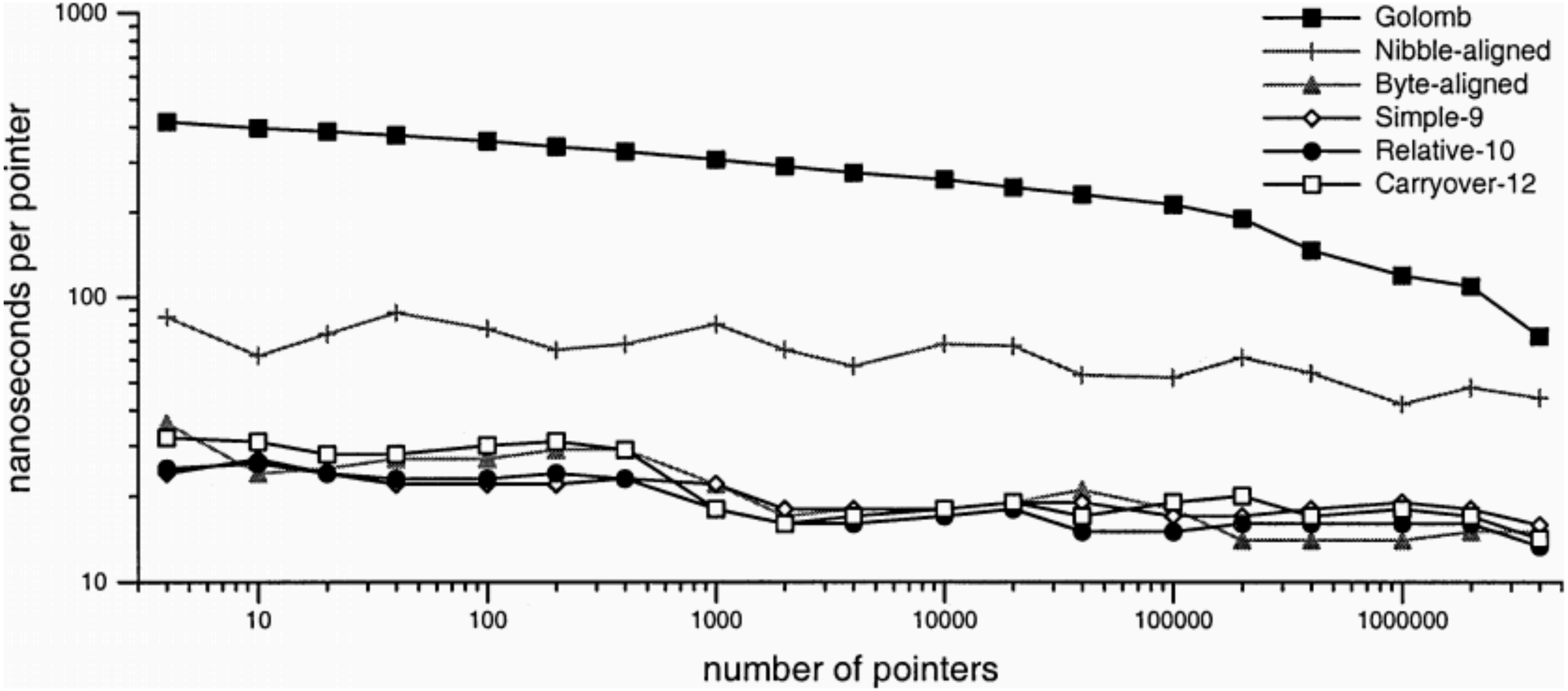
Carryover-12: two more tweaks

- Similarly, if we’re in the situation where 32 bits are being used for data, but we allocate two 15 bit codes, then we have 2 bits free for the next selector
- Likewise, five 6 bits codes (in the 32-bit situation) leaves room for the selector of the next word
- …and there are other combinations.
- i.e., in Carryover-12, each word contains either
  - 32 data bits, if the selector can be fitter in previous word, or
  - A selector + 30 data bits if it cannot

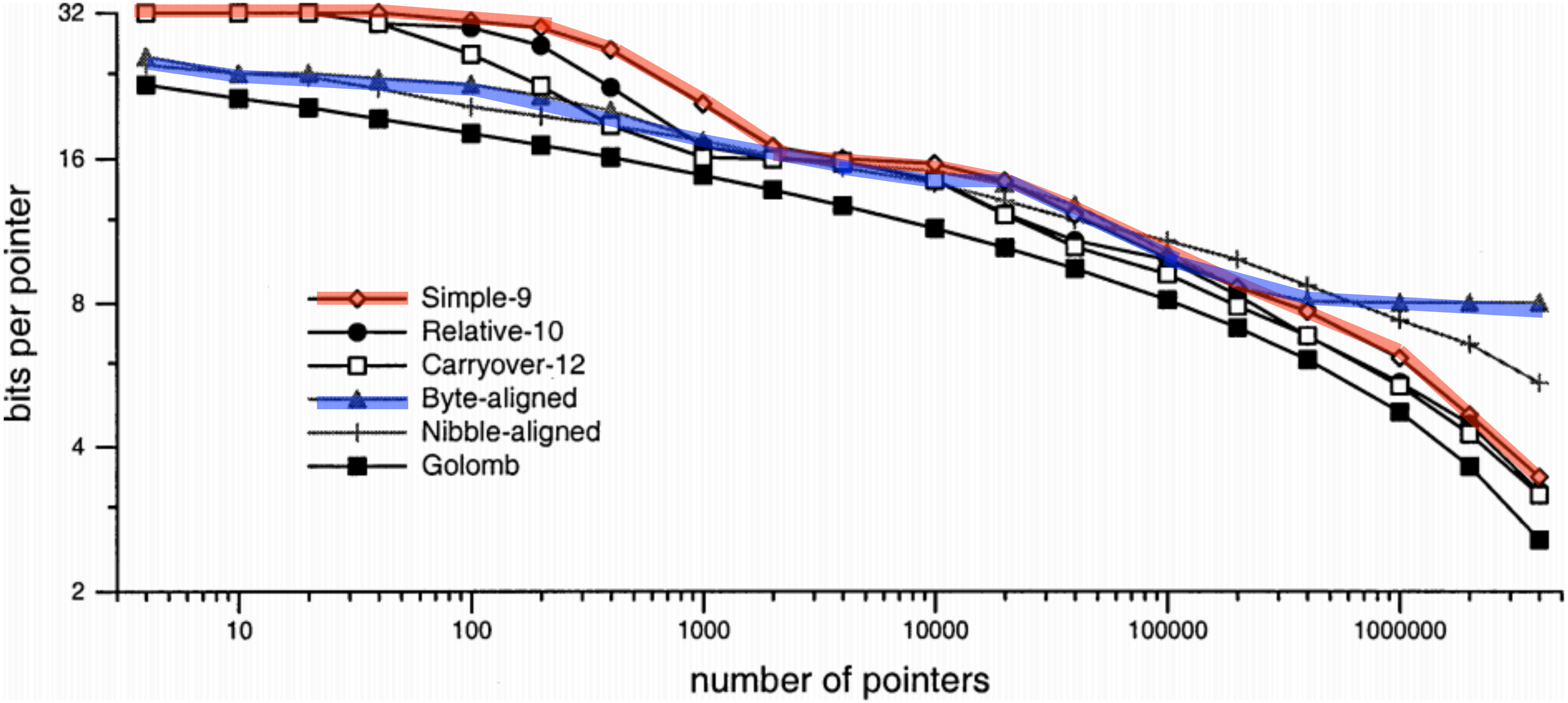
Relative-10: Compression Performance



Decoding speed



Relative-10: Compression Performance



Summary

- Integer codes are used in many many high-performance industrial software systems
  - Whenever throughput is important they can usually be found.

- Having integer codes respect memory alignment can lead to significant speed savings, usually at the cost of some loss in compression, but even this can be kept quite small, as we have seen.

- We will encounter integer codes again when we look at compressed data structures.

End

Next lecture…

13/01	Shannon’s Theorem
15/01	Huffman Coding
20/01	Integer Codes I
22/01	Integer Codes II
27/01	Dynamic Prefix Coding
29/01	Arithmetic Coding
03/02	Text Compression
05/02	*No Lecture*
10/02	Text Compression
12/02	Compressed Data Structures
17/02	Compressed Data Structures
19/02	Compressed Data Structures
24/02	Compressed Data Structures