Techniques for Inverted Index Compression

Giulio Ermanno Pibiri, ISTI-CNR
Rossano Venturini, University of Pisa

6.506 Algorithm Engineering – Paper Presentation

Presenter: Joseph Zhang

Problem and Background

Inverted Index – Example

```
-  Document 0 : here are some terms
-  Document 1 : some more terms
-  Document 2 : even more terms
```

Inverted Index:

```
-  here  : 0
-  are   : 0
-  some  : 0, 1
-  terms : 0, 1, 2
-  more  : 1, 2
-  even  : 2
```

Inverted Index

For each term t in collection of documents, we want to keep track of the documents where the term appears in inverted lists

Wide range of applications, for example:

- large-scale search engines
- social networks
- data storage architectures
- database searching

Many indexed documents and heavy query loads – compression can help!

Inverted Index Compression

Inverted Index Compression – Timeline

1949	Shannon-Fano [32, 93]
1952	Huffman [43]
1963	Arithmetic [1][1]
1966	Golomb [40]
1971	Elias-Fano [30, 33]; Rice [87]
1972	Variable-Byte and Nibble [101]
1975	Gamma and Delta [31]
1978	Exponential Golomb [99]
1985	Fibonacci-based [6, 37]
1986	Hierarchical bit-vectors [35]
1988	Based on Front Coding [16]
1996	Interpolative [65, 66]
1998	Frame-of-Reference (For) [39]; modified Rice [2]
2003	SC-dense [11]
2004	Zeta [8, 9]

2005	Simple-9, Relative-10, and Carryover-12 [3]; RBUC [60]
2006	PForDelta [114]; BASC [61]
2008	Simple-16 [112]; Tournament [100]
2009	ANS [27]; Varint-GB [23]; Opt-PFor [111]
2010	Simple8b [4]; VSE [96]; SIMD-Gamma [91]
2011	Varint-G8IU [97]; Parallel-PFor [5]
2013	DAC [12]; Quasi-Succinct [107]
2014	Partitioned Elias-Fano [73]; QMX [103]; Roaring [15, 51, 53]
2015	BP32, SIMD-BP128, and SIMD-FastPFor [50]; Masked-VByte [84]
2017	Clustered Elias-Fano [80]
2018	Stream-VByte [52]; ANS-based [63, 64]; Opt-VByte [83]; SIMD-Delta [104]; general-purpose compression libraries [77]
2019	DINT [79]; Slicing [78]

Goals

Survey of encoding algorithms useful for inverted index compression

- hierarchical division in three main classes

Characterize performance of inverted index through experimentation

- compression effectiveness
- query operation performance
- sequential decoding speed

Inverted Index Compression – Organization

Techniques organized into three main classes:

Integer Codes: algorithms that compress a single integer

List Compressors: algorithms that compress lists of many integers together

Index Compressors: algorithms that represent many lists together

Integer Codes

Overview

Assign each integer x a uniquely decodable variable length code

- want to be able to decode without ambiguity from left to right

Ideal codeword length for integer x is log2(1/P(x)) bits

- we can derive optimal distribution for an encoding from this

Prefix-Free Codes

No codeword is a prefix of another codeword

Lexicographic ordering: codewords in same order as the integers, and this can help us speed up the encoding and decoding process

Table 2. Example Prefix-Free Code for the Integers 1..8, Along with Associated Codewords, Codeword Lengths, and Corresponding Left-Justified, 7-Bit Integers

(a)

x	Codewords	*Lengths*	*Values*
1	`0`	1	0
2	`100`	3	64
3	`101`	3	80
4	`11000`	5	96
5	`11001`	5	100
6	`11010`	5	104
7	`11011`	5	108
8	`1110000`	7	112
–	–	–	127

(b)

Lengths	*First*	*Values*
1	1	0
2	2	64
3	2	64
4	4	96
5	4	96
6	8	112
7	8	112
–	9	127

The codewords are left-justified to better highlight their lexicographic order. In (b), the compact version of the table in (a), used by the encoding/decoding procedures coded in Figure 1. The “values” and “first” columns are padded with a sentinel value (in gray) to let the search be well defined.

Prefix-Free Codes

Encode(x) :

- determine ℓ such that $\mathit{first}[\ell] \leq x < \mathit{first}[\ell + 1]$
- $\mathit{offset} = x - \mathit{first}[\ell]$
- $\mathit{jump} = 1 \ll (M - \ell)$
- $\mathrm{Write}((\mathit{values}[\ell] + \mathit{offset} \times \mathit{jump}) \gg (M - \ell), \ell)$

Decode() :

- determine ℓ such that $\mathit{values}[\ell] \leq \mathit{buffer} < \mathit{values}[\ell + 1]$
- $\mathit{offset} = (\mathit{buffer} - \mathit{values}[\ell]) \gg (M - \ell)$
- $\mathit{buffer} = ((\mathit{buffer} \ll \ell)\ \&\ \mathrm{MASK}) + \mathrm{Take}(\ell)$ ▹ MASK is the constant $2^M - 1$.
- **return** $\mathit{first}[\ell] + \mathit{offset}$

Fig. 1. Encoding and decoding procedures using two parallel arrays *first* and *values* of $M + 1$ values each.

Unary Coding

x-1 ones followed by a single zero

length is x

optimal when

$P(x) = 2^{-x}$

Table 3. Integers 1..8 as Represented with Several Codes

x	$U(x)$	$B(x)$	$\gamma(x)$	$\delta(x)$	$G_2(x)$	$ExpG_2(x)$	$Z_2(x)$
1	0	0	0.	0.	0.0	0.00	0.0
2	10	1	10.0	100.0	0.1	0.01	0.10
3	110	10	10.1	100.1	10.0	0.10	0.11
4	1110	11	110.00	101.00	10.1	0.11	10.000
5	11110	100	110.01	101.01	110.0	10.000	10.001
6	111110	101	110.10	101.10	110.1	10.001	10.010
7	1111110	110	110.11	101.11	1110.0	10.010	10.011
8	11111110	111	1110.000	11000.000	1110.1	10.011	10.1000

The "." symbol highlights the distinction between different parts of the codes and has a purely illustrative purpose: it is not included in the final coded representation.

Binary Coding

representation of x-1 in binary

length is k, where the integers are bounded by 2^k

optimal when

P(x) = 2^-k

Table 3. Integers 1..8 as Represented with Several Codes

x	$\mathrm{U}(x)$	$\mathrm{B}(x)$	$\gamma(x)$	$\delta(x)$	$\mathrm{G}_2(x)$	$\mathrm{ExpG}_2(x)$	$\mathrm{Z}_2(x)$
1	0	0	0.	0.	0.0	0.00	0.0
2	10	1	10.0	100.0	0.1	0.01	0.10
3	110	10	10.1	100.1	10.0	0.10	0.11
4	1110	11	110.00	101.00	10.1	0.11	10.000
5	11110	100	110.01	101.01	110.0	10.000	10.001
6	111110	101	110.10	101.10	110.1	10.001	10.010
7	1111110	110	110.11	101.11	1110.0	10.010	10.011
8	11111110	111	1110.000	11000.000	1110.1	10.011	10.1000

The "." symbol highlights the distinction between different parts of the codes and has a purely illustrative purpose: it is not included in the final coded representation.

Gamma Coding

U(|bin(x)|) followed by last |bin(x)-1| bits from bin(x)

length is 2|bin(x)|-1

optimal when

P(x) = 1/2k^2

Table 3. Integers 1..8 as Represented with Several Codes

x	$U(x)$	$B(x)$	$\gamma(x)$	$\delta(x)$	$G_2(x)$	$ExpG_2(x)$	$Z_2(x)$
1	0	0	0.	0.	0.0	0.00	0.0
2	10	1	10.0	100.0	0.1	0.01	0.10
3	110	10	10.1	100.1	10.0	0.10	0.11
4	1110	11	110.00	101.00	10.1	0.11	10.000
5	11110	100	110.01	101.01	110.0	10.000	10.001
6	111110	101	110.10	101.10	110.1	10.001	10.010
7	1111110	110	110.11	101.11	1110.0	10.010	10.011
8	11111110	111	1110.000	11000.000	1110.1	10.011	10.1000

The "." symbol highlights the distinction between different parts of the codes and has a purely illustrative purpose: it is not included in the final coded representation.

Delta Coding

γ(|bin(x)|) followed by last |bin(x)-1| bits from bin(x)

length is |γ (|bin(x)|)|+|bin(x)|-1

optimal when P(x) = 1/(2x(log2(x)^2)

k-gamma, SIMD delta codes

Table 3. Integers 1..8 as Represented with Several Codes

x	$U(x)$	$B(x)$	$\gamma(x)$	$\delta(x)$	$G_2(x)$	$ExpG_2(x)$	$Z_2(x)$
1	0	0	0.	0.	0.0	0.00	0.0
2	10	1	10.0	100.0	0.1	0.01	0.10
3	110	10	10.1	100.1	10.0	0.10	0.11
4	1110	11	110.00	101.00	10.1	0.11	10.000
5	11110	100	110.01	101.01	110.0	10.000	10.001
6	111110	101	110.10	101.10	110.1	10.001	10.010
7	1111110	110	110.11	101.11	1110.0	10.010	10.011
8	11111110	111	1110.000	11000.000	1110.1	10.011	10.1000

The "." symbol highlights the distinction between different parts of the codes and has a purely illustrative purpose: it is not included in the final coded representation.

Golomb

unary encoding of quotient then minimal binary of remainder

optimal when

P(x) = p(1-p)^(x-1)

gaps between integers drawn at random follows geometric distribution

Table 3. Integers 1..8 as Represented with Several Codes

x	$\mathrm{U}(x)$	$\mathrm{B}(x)$	$\gamma(x)$	$\delta(x)$	$\mathrm{G}_2(x)$	$\mathrm{ExpG}_2(x)$	$\mathrm{Z}_2(x)$
1	0	0	0.	0.	0.0	0.00	0.0
2	10	1	10.0	100.0	0.1	0.01	0.10
3	110	10	10.1	100.1	10.0	0.10	0.11
4	1110	11	110.00	101.00	10.1	0.11	10.000
5	11110	100	110.01	101.01	110.0	10.000	10.001
6	111110	101	110.10	101.10	110.1	10.001	10.010
7	1111110	110	110.11	101.11	1110.0	10.010	10.011
8	11111110	111	1110.000	11000.000	1110.1	10.011	10.1000

The "." symbol highlights the distinction between different parts of the codes and has a purely illustrative purpose: it is not included in the final coded representation.

Rice

special case of Golomb with b=2^k

remainder written in k bits, so length is floor((x-1)/2^k)+k+1

Table 3. Integers 1..8 as Represented with Several Codes

x	$\mathrm{U}(x)$	$\mathrm{B}(x)$	$\gamma(x)$	$\delta(x)$	$\mathrm{G}_2(x)$	$\mathrm{ExpG}_2(x)$	$\mathrm{Z}_2(x)$
1	0	0	0.	0.	0.0	0.00	0.0
2	10	1	10.0	100.0	0.1	0.01	0.10
3	110	10	10.1	100.1	10.0	0.10	0.11
4	1110	11	110.00	101.00	10.1	0.11	10.000
5	11110	100	110.01	101.01	110.0	10.000	10.001
6	111110	101	110.10	101.10	110.1	10.001	10.010
7	1111110	110	110.11	101.11	1110.0	10.010	10.011
8	11111110	111	1110.000	11000.000	1110.1	10.011	10.1000

The "." symbol highlights the distinction between different parts of the codes and has a purely illustrative purpose: it is not included in the final coded representation.

Exponential Golomb

Unary encoding of bucket identifier followed by binary encoding of bucket offset

$$B = \left[0, 2^k, \sum_{i=0}^{1} 2^{k+i}, \sum_{i=0}^{2} 2^{k+i}, \sum_{i=0}^{3} 2^{k+i}, \ldots\right]$$

Zeta

Exponential Golomb with these buckets:

$$[0, 2^k - 1, 2^{2k} - 1, 2^{3k} - 1, \ldots]$$

optimal for power law distribution with small exponent, i.e.

P(x) = 1/(ζ(α)x^α) where ζ is Riemann zeta function

Fibonacci

every positive integer has unique representation as sum of some non-adjacent Fibonacci numbers

1 for if i-th Fibonacci number used in sum, 0 otherwise, final control 1 bit

optimal when P(x) is approximately 1/(2x^1.44)

can generate lexicographic codewords from the lengths

Table 4. Integers 1..8 as Represented with Fibonacci-Based Codes

(a) "Original" Codewords

x	F(x)					
1	1	**1**				
2	0	1	**1**			
3	0	0	1	**1**		
4	1	0	1	**1**		
5	0	0	0	1	**1**	
6	1	0	0	1	**1**	
7	0	1	0	1	**1**	
8	0	0	0	0	1	**1**
F_i	1	2	3	5	8	13

(b) Lexicographic Codewords

x	F(x)					
1	0	0				
2	0	1	0			
3	0	1	1	0		
4	0	1	1	1		
5	1	0	0	0	0	
6	1	0	0	0	1	
7	1	0	0	1	0	
8	1	0	0	1	1	0

In (a), the final control bit is highlighted in bold font and the relevant Fibonacci numbers F_i involved in the representation are also shown at the bottom of the table. In (b), the "canonical" lexicographic codewords are presented.

Variable-Byte

Codes previously described are bit-aligned, but byte or word-aligned codes can have better decoding speed

MSB signals continuation of byte sequence, rest of the bits used for data

Optimal when P(x) approximately x^(-8/7) for byte-aligned

Example: 00000100.10000001.11111110

Branch Prediction

Varint-GB groups control bits together to reduce the probability of a branch misprediction for higher throughput

Assume largest represented integer fits in 4 bytes, then we have only four different byte-lengths, which only requires two bits

So groups of four such integers requires only one control byte

SIMD Parallelism

Varint-G8IU: one control byte for variable number of integers in 8-byte segment, for groups of between two and eight compressed integers

Masked-VByte: decoding by first gathering MSBs of consecutive bytes with SIMD instruction, and permuting data bytes accordingly

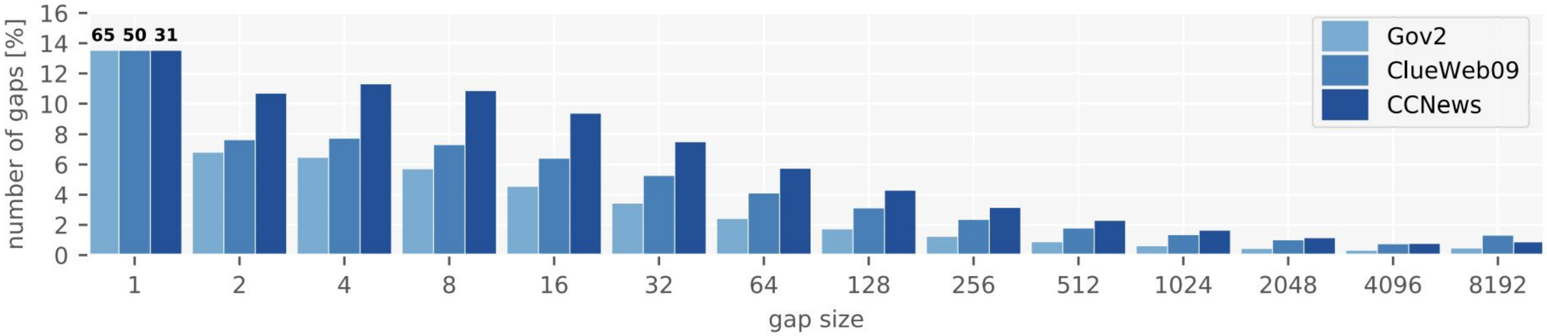
Stream-VByte: separate encoding of control data bits into separate streams, allows decoding multiple control bits separately and reduces data dependencies

SC-Dense

Variable-Byte has 2^7 as separator between stoppers and continuers, we can generalize this to adapt to distribution in question.

Table 5. Integers 1..20 as Represented by SC(4,4)- and SC(5,3)-Dense Codes, Respectively

x	SC$(4,4,x)$	SC$(5,3,x)$	x	SC$(4,4,x)$	SC$(5,3,x)$
1	000	000	11	101.010	110.000
2	001	001	12	101.011	110.001
3	010	010	13	110.000	110.010
4	011	011	14	110.001	110.011
5	100.000	100	15	110.010	110.100
6	100.001	101.000	16	110.011	111.000
7	100.010	101.001	17	111.000	111.001
8	100.011	101.010	18	111.001	111.010
9	101.000	101.011	19	111.010	111.011
10	101.001	101.100	20	111.011	111.100

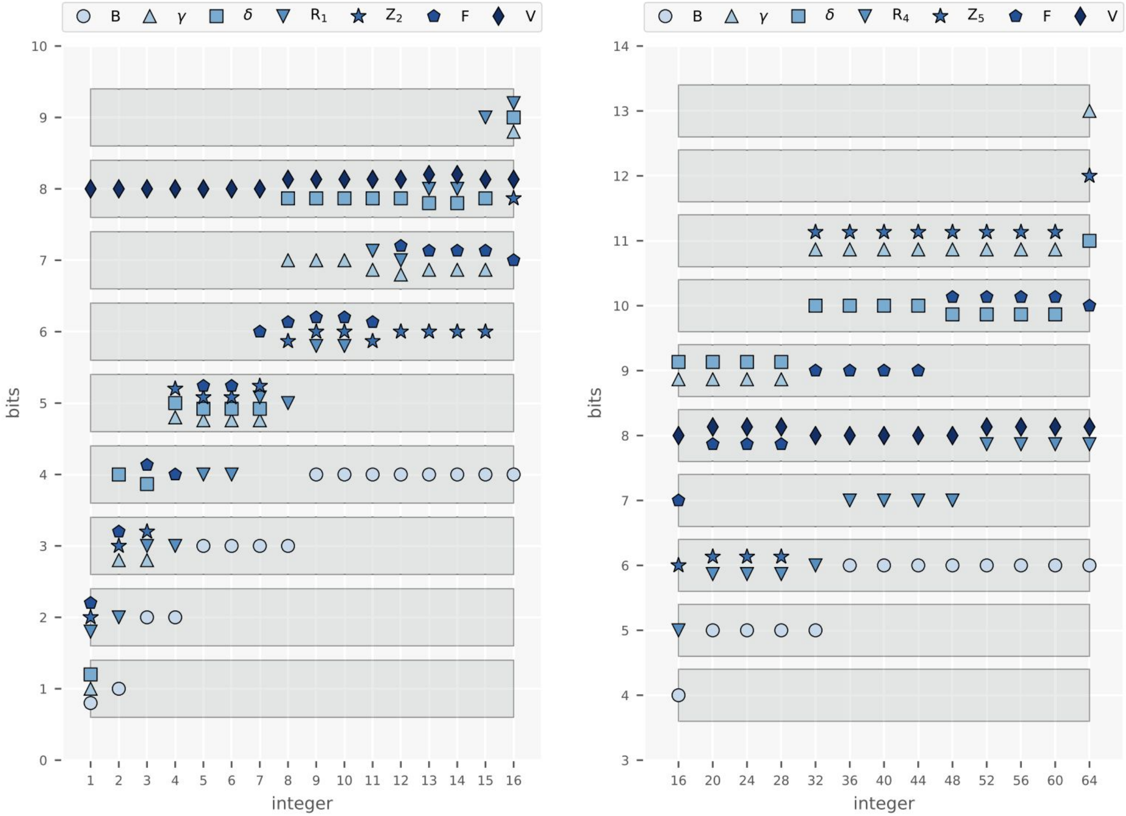


Summary

For our inverted indexes with sorted inverted lists, compression on gaps of the sequence works well

Tuned parametric codes can be good, but tuning not always possible

Some different relative strengths at smaller and larger values



List Compressors

Overview

Information-theoretic lower bound gives minimum bits to represent list of n strictly increasing integers drawn at random from a universe

In practice, compressors often take advantage of inverted lists having clusters of close integers, which are more compressible, to use less than the bound

These can arise because of indexed documents often being clustered by sharing the same set of terms

Binary Packing

Partition sequence into blocks of fixed or variable length and encode them separately

- if sequence has clusters of close integers, values are likely to be of similar magnitude

Compute bit width of max element in block and represent integers in block with that number of bits, gaps between integers can be computed to lower width

- variable sized blocks usually preferable

Many variants of this approach, including word-aligned version

Simple

split sequence into fixed-memory units, and pack as many integers as we can fit into a unit

selector code gives information on how elements are packed in segment

typically provides good decoding speed and good compression

Table 6. Nine Different Ways of Packing Integers in a 28-Bit Segment as Used by Simple9

4-Bit Selector	Integers	Bits per Integer	Wasted Bits
0000	28	1	0
0001	14	2	0
0010	9	3	1
0011	7	4	0
0100	5	5	3
0101	4	7	0
0110	3	9	1
0111	2	14	0
1000	1	28	0

PForDelta

Space inefficiency of block-based strategies like simple when there is just one large value in the block

"patched" frame of reference chooses a range that fits most of the integers, and encodes exceptions separately with different algorithm

$$[3, 4, 7, 21, 9, 12, 5, 16, 6, 2, 34]$$

$$[1, 2, 5, *, 7, 10, 3, *, 4, 0, *] - [21, 16, 34]$$

Elias-Fano

For n sorted integers in range [1, U]

Split them into l=floor(log2(U/n)) low bits and ceil(log2(U))-l high bits

Low bits are encoded separately with nl size bitvector directly

High bits are encoded separately with 2n bits: for high bit h_i, we set the bit in position h_i+i, so unary encoding of how many integers have h_i equal to value

Elias-Fano – Example

Table 7. Example of Elias-Fano Encoding Applied to the Sequence
$\mathcal{S} = [3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54, 62]$

$\mathcal{S}$	3	4	7	13	14	15	21	25	36	38		54	62
	0	0	0	0	0	0	0	0	1	1	1	1	1
high	0	0	0	0	0	0	1	1	0	0	0	1	1
	0	0	0	1	1	1	0	1	0	0	1	0	1
	0	1	1	1	1	1	1	0	1	1		1	1
low	1	0	1	0	1	1	0	0	0	1		1	1
	1	0	1	1	0	1	1	1	0	0		0	0
H	1110			1110			10	10	110		0	10	10
L	011.100.111			101.110.111			101	001	100.110			110	110

Elias-Fano – Random Accesses

How to decode an individual integer S[i] in O(1) time:

- need data structure to get i-th bit set to b in H (high bits bitvector) in O(1), it turns out this only requires o(n) bits, small compared to encoding size
- call the query above Select_b(i), then high bits is Select_1(i)-i, since we unary encoded how many integers share same high part, so there is 1 for each integer and 0 for each distinct high part
- read low bits directly from L (low bits bitvector)
- concatenate high and low bits

Elias-Fano – Successor Queries

How to find smallest integer at least x, for some integer x:

- let h_x be high bits of x
- i = Select_0(h_x)-h_x+1 (for h_x>0, use i=0 otherwise) says that there are i integers in S with high bits less than h_x
- j = Select_0(h_x+1)-h_x is start position for elements with larger high bits
- then only the range from i to j needs to be searched
- runs in O(1+log(U/n)) time
- this successor query is also called NextGEQ

Elias Fano – Partitioning

High and low bit split can be chosen arbitrarily, for a non-parametric split

Roaring partitions U <= 2^32 into chunks of 2^16 values each, and encodes chunks depending on if they are sparse, dense, or very dense

- sorted array for sparse, bitmap for dense, runs for very dense

Slicing also continues encoding recursively for the sparse chunks

Main idea in these is to find dense regions for bitmap encodings but treat the sparse regions differently

Interpolative

Binary Interpolative Code (BIC)

Recursively split list in half and encode middle element with minimal bits

Fully make use of runs of consecutive integers

Interpolative – Example

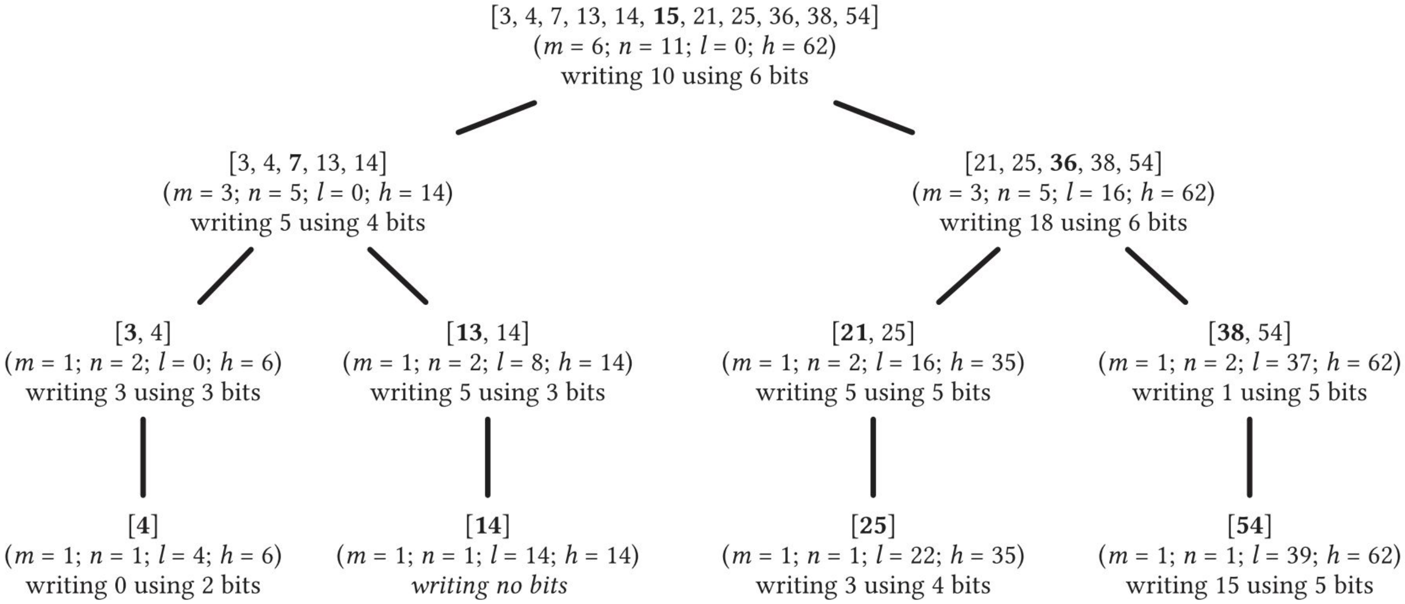


Fig. 4. The recursive calls performed by the Binary Interpolative Coding algorithm when applied to the sequence [3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54] with initial knowledge of lower and upper bound values $l = 0$ and $h = 62$. In bold font, we highlight the middle element being encoded.

Directly-Addressable Codes

Reduce problem of random access to ranking over a bitmap

Hybrid Approaches

Hybrid approaches can use different compressors for blocks of a list

Example: collect access statistics for blocks

- represent rarely accessed blocks with a more space-efficient compressor
- represent frequently accessed blocks with a more time-efficient compressor

Some algorithms for optimal partitioning into blocks for these hybrid approaches

Entropy Coding

Usually not as competitive for efficiency and simplicity of implementation

Huffman, Arithmetic, and Asymmetric Numeral Systems (ANS)

Index Compressors

Clustered

Inverted lists grouped into clusters of lists sharing as many integers as possible

For each cluster, we have a reference list, where for integers in reference list and list in cluster, they can be represented as position occupied in reference list

Any compressor can be used for the intersections between the reference list and cluster lists but PEF (partitioned Elias-Fano) was used by authors

Time/space tradeoffs from varying size of the reference lists

ANS Based

Alphabet size of ANS method may be too large for representing our integers even if we only work with gaps

VByte+ANS: this can be adapted by preprocessing with Variable-Byte to reduce input list, and then applying ANS on the sequence of bytes

Dictionary Based

Store most frequent 2^b patterns in dictionary, and use it to encode subsequences of gaps, as there are often very repetitive patterns across the whole inverted index

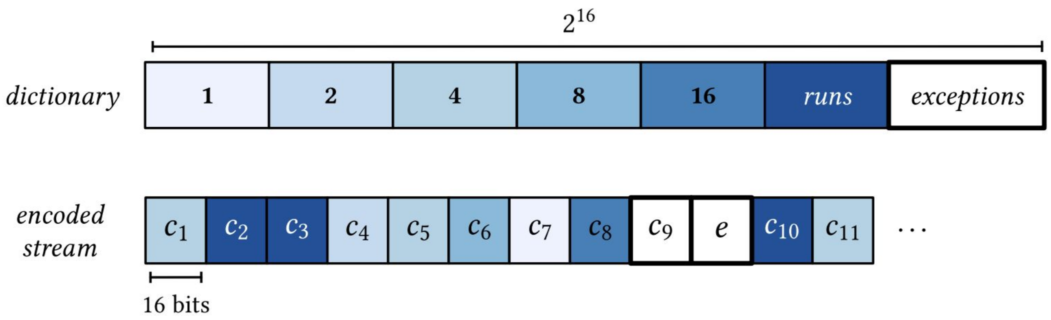


Fig. 6. A dictionary-based encoded stream example, where dictionary entries corresponding to $\{1, 2, 4, 8, 16\}$-long integer patterns, runs, and exceptions are labeled with different shades. Once provision has been made for such a dictionary structure, a sequence of gaps can be modeled as a sequence of codewords $\{c_k\}$, each being a reference to a dictionary entry, as represented with the *encoded stream* in the picture. Note that, for example, codeword c_9 signals an exception, and therefore the next symbol e is decoded using an escape mechanism.

Experiments

Experimental Setup

Mainly focused on comparing some selected representations over being completely exhaustive, some further comparisons in their benchmark repository

Table 9. Different Tested Index Representations

Method	Partitioned by	SIMD	Alignment	Description
VByte	Cardinality	Yes	Byte	Fixed-size partitions of 128
Opt-VByte	Cardinality	Yes	Bit	Variable-size partitions
BIC	Cardinality	No	Bit	Fixed-size partitions of 128
δ	Cardinality	No	Bit	Fixed-size partitions of 128
Rice	Cardinality	No	Bit	Fixed-size partitions of 128
PEF	Cardinality	No	Bit	Variable-size partitions
DINT	Cardinality	No	16-bit word	Fixed-size partitions of 128
Opt-PFor	Cardinality	No	32-bit word	Fixed-size partitions of 128
Simple16	Cardinality	No	64-bit word	Ffixed-size partitions of 128
QMX	Cardinality	Yes	128-bit word	Fixed-size partitions of 128
Roaring	Universe	Yes	byte	Single span
Slicing	Universe	Yes	byte	Multi-span

Table 10. Datasets Used in the Experiments

(a) Basic Statistics

	Gov2	ClueWeb09	CCNews
Lists	39,177	96,722	76,474
Universe	24,622,347	50,131,015	43,530,315
Integers	5,322,883,266	14,858,833,259	19,691,599,096
Entropy of the gaps	3.02	4.46	5.44
$\lceil \log_2 \rceil$ of the gaps	1.35	2.28	2.99

(b) TREC 2005/06 Queries

	Gov2	ClueWeb09	CCNews
Queries	34,327	42,613	22,769
2 terms	32.2%	33.6%	37.5%
3 terms	26.8%	26.5%	27.3%
4 terms	18.2%	17.7%	16.8%
5+ terms	22.8%	22.2%	18.4%

Compression Effectiveness

Table 11. Space Effectiveness in Total GiB and Bits per Integer, and Nanoseconds per Decoded Integer

Method	Gov2			ClueWeb09			CCNews		
	GiB	Bits/int	ns/int	GiB	Bits/int	ns/int	GiB	bits/int	ns/int
VByte	5.46	8.81	0.96	15.92	9.20	1.09	21.29	9.29	1.03
Opt-VByte	2.41	3.89	0.73	9.89	5.72	0.92	14.73	6.42	0.72
BIC	1.82	2.94	5.06	7.66	4.43	6.31	12.02	5.24	6.97
δ	2.32	3.74	3.56	8.95	5.17	3.72	14.58	6.36	3.85
Rice	2.53	4.08	2.92	9.18	5.31	3.25	13.34	5.82	3.32
PEF	1.93	3.12	0.76	8.63	4.99	1.10	12.50	5.45	1.31
DINT	2.19	3.53	1.13	9.26	5.35	1.56	14.76	6.44	1.65
Opt-PFor	2.25	3.63	1.38	9.45	5.46	1.79	13.92	6.07	1.53
Simple16	2.59	4.19	1.53	10.13	5.85	1.87	14.68	6.41	1.89
QMX	3.17	5.12	0.80	12.60	7.29	0.87	16.96	7.40	0.84
Roaring	4.11	6.63	0.50	16.92	9.78	0.71	21.75	9.49	0.61
Slicing	2.67	4.31	0.53	12.21	7.06	0.68	17.83	7.78	0.69

Sequential Decoding

Table 11. Space Effectiveness in Total GiB and Bits per Integer, and Nanoseconds per Decoded Integer

Method	Gov2			ClueWeb09			CCNews		
	GiB	Bits/int	ns/int	GiB	Bits/int	ns/int	GiB	bits/int	ns/int
VByte	5.46	8.81	0.96	15.92	9.20	1.09	21.29	9.29	1.03
Opt-VByte	2.41	3.89	0.73	9.89	5.72	0.92	14.73	6.42	0.72
BIC	1.82	2.94	5.06	7.66	4.43	6.31	12.02	5.24	6.97
δ	2.32	3.74	3.56	8.95	5.17	3.72	14.58	6.36	3.85
Rice	2.53	4.08	2.92	9.18	5.31	3.25	13.34	5.82	3.32
PEF	1.93	3.12	0.76	8.63	4.99	1.10	12.50	5.45	1.31
DINT	2.19	3.53	1.13	9.26	5.35	1.56	14.76	6.44	1.65
Opt-PFor	2.25	3.63	1.38	9.45	5.46	1.79	13.92	6.07	1.53
Simple16	2.59	4.19	1.53	10.13	5.85	1.87	14.68	6.41	1.89
QMX	3.17	5.12	0.80	12.60	7.29	0.87	16.96	7.40	0.84
Roaring	4.11	6.63	0.50	16.92	9.78	0.71	21.75	9.49	0.61
Slicing	2.67	4.31	0.53	12.21	7.06	0.68	17.83	7.78	0.69

Boolean AND Queries

Table 12. Milliseconds Spent per AND Query by Varying the Number of Query Terms

Method	Gov2					ClueWeb09					CCNews				
	2	3	4	5+	avg.	2	3	4	5+	avg.	2	3	4	5+	avg.
VByte	2.2	2.8	2.7	3.3	2.8	10.2	12.1	13.7	13.9	12.5	14.0	22.4	19.7	21.9	19.5
Opt-VByte	2.8	3.1	2.8	3.2	3.0	12.2	13.3	14.0	13.6	13.3	16.0	23.2	19.6	20.3	19.8
BIC	6.8	9.7	10.4	13.2	10.0	31.7	44.2	51.5	53.8	45.3	45.6	79.7	76.9	88.8	72.8
δ	4.6	6.3	6.5	8.2	6.4	20.9	28.3	33.5	34.5	29.3	28.6	50.9	48.0	55.6	45.8
Rice	4.1	5.6	5.8	7.3	5.7	19.2	25.7	30.2	31.1	26.6	26.5	46.5	43.5	50.1	41.6
PEF	2.5	3.1	2.8	3.2	2.9	12.3	13.5	14.4	13.8	13.5	17.2	24.6	21.0	21.9	21.2
DINT	2.5	3.3	3.3	4.1	3.3	11.9	14.6	16.5	17.1	15.0	16.9	27.3	24.6	28.1	24.2
Opt-PFor	2.6	3.5	3.5	4.3	3.5	12.8	15.9	18.0	18.3	16.3	16.6	27.2	24.3	27.1	23.8
Simple16	2.8	3.7	3.7	4.6	3.7	12.8	16.3	18.4	18.9	16.6	17.6	28.8	26.3	29.5	25.5
QMX	2.0	2.6	2.5	3.0	2.5	9.6	11.5	13.0	13.1	11.8	13.3	21.5	18.8	20.8	18.6
Roaring	0.3	0.5	0.7	0.8	0.6	1.5	2.5	3.1	4.3	2.9	1.1	2.0	2.6	4.1	2.5
Slicing	0.3	1.0	1.2	1.6	1.0	1.5	4.5	5.4	6.7	4.5	1.8	4.3	5.1	6.0	4.3

Boolean OR Queries

Table 13. Milliseconds Spent per OR Query by Varying the Number of Query Terms

Method	Gov2					ClueWeb09					CCNews				
	2	3	4	5+	avg.	2	3	4	5+	avg.	2	3	4	5+	avg.
VByte	6.8	24.4	54.7	131.7	54.4	20.1	71.3	156.0	379.5	156.7	24.4	94.5	178.8	391.4	172.3
Opt-VByte	11.0	35.7	77.4	176.0	75.0	31.3	101.4	213.4	500.1	211.6	36.4	128.0	232.0	510.4	226.7
BIC	16.7	50.3	105.0	238.8	102.7	49.9	145.3	290.4	668.2	288.4	64.4	193.8	332.6	692.5	320.8
δ	12.6	40.8	87.9	202.5	85.9	34.9	112.9	236.7	557.7	235.6	42.2	144.9	263.8	571.3	255.5
Rice	13.4	43.1	93.3	211.3	90.3	36.8	118.2	248.5	576.6	245.0	43.6	149.3	270.5	585.6	262.2
PEF	10.2	33.0	71.7	164.2	69.8	31.1	99.7	208.5	492.3	207.9	37.6	127.5	232.6	507.1	226.2
DINT	8.5	28.5	63.7	147.6	62.1	24.9	84.1	178.8	424.3	178.0	30.6	109.2	200.4	432.7	193.2
Opt-PFor	8.9	31.1	69.4	161.4	67.7	27.0	90.8	194.0	453.5	191.3	31.3	113.2	209.0	447.2	200.2
Simple16	7.8	26.2	58.3	138.2	57.6	23.7	78.0	165.5	394.7	165.5	28.7	101.5	185.3	397.8	178.4
QMX	6.6	23.8	53.4	128.1	53.0	19.7	70.0	153.2	377.9	155.2	24.0	92.6	175.2	382.4	168.6
Roaring	1.2	2.8	4.3	6.4	3.7	4.7	9.0	12.0	15.7	10.3	3.8	7.6	10.5	15.1	9.2
Slicing	1.3	4.0	6.3	9.2	5.2	5.0	12.8	18.1	25.3	15.3	5.8	12.9	17.3	23.0	14.8

Space/Time Tradeoffs

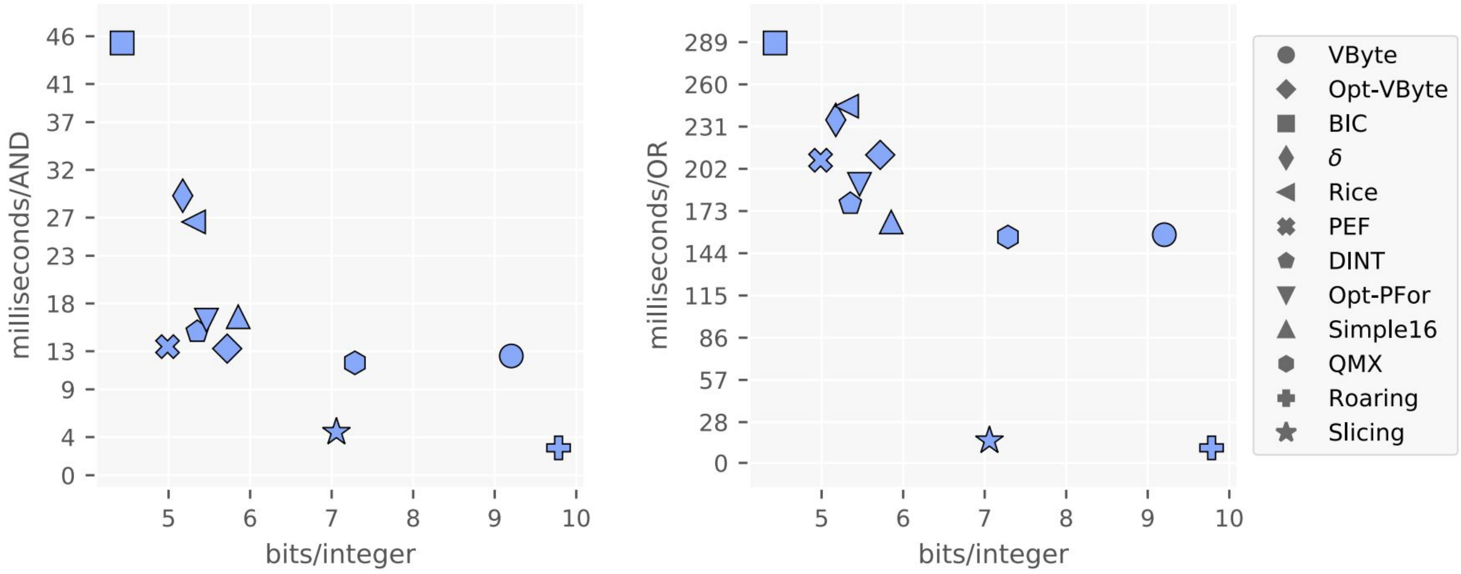


Fig. 7. Space/time trade-off curves for the ClueWeb09 dataset.

Future Work

Simpler compression formats that can be decoded faster using low-latency instructions and minimal branches

Making full use of superscalar execution and SIMD instructions

Dynamic compressed representations for integer sequences that can support additions and deletions, a specific case of more general dictionary problem

Implementations with good practical performance

Techniques for Inverted Index Compression

Giulio Ermanno Pibiri, ISTI-CNR
Rossano Venturini, University of Pisa