

POSTINGS COMPRESSION

We introduce the topic of integer compression considering the important example of Posting Lists compression

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key desideratum: store each posting compactly.
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $\log_2 800,000 \approx 20$ bits per docID.
- Our goal: use far fewer than 20 bits per docID.

Postings: two conflicting forces

- A term like *arachnocentric* occurs in maybe one doc out of a million – we would like to store this posting using $\log_2 1M \sim 20$ bits.
- A term like *the* occurs in virtually every doc, so 20 bits/posting is too expensive.
 - Prefer 0/1 bitmap vector in this case

Postings file entry

- We store the list of docs containing a term in increasing order of docID.
 - **computer**: 33,47,154,159,202 ...
- Consequence: it suffices to store *gaps*.
 - 33,14,107,5,43 ...
- Hope: most gaps can be encoded/stored with far fewer than 20 bits.

Three postings entries

	encoding	postings list				
THE	docIDs	...	283042	283043	283044	283045 ...
	gaps		1	1	1	...
COMPUTER	docIDs	...	283047	283154	283159	283202 ...
	gaps		107	5	43	...
ARACHNOCENTRIC	docIDs	252000	500100			
	gaps	252000	248100			

Variable length encoding

- Aim:
 - For *arachnocentric*, we will use ~20 bits/gap entry.
 - For *the*, we will use ~1 bit/gap entry.
- If the average gap for a term is G , we want to use $\sim \log_2 G$ bits/gap entry.
- Key challenge: encode every integer (gap) with about as few bits as needed for that integer.
- This requires a *variable length encoding*
- Variable length codes achieve this by using short codes for small numbers

Variable Byte (VB) codes

- For a gap value G , we want to use close to the fewest bytes needed to hold $\log_2 G$ bits
- Begin with one byte to store G and dedicate 1 bit in it to be a continuation bit c
- If $G \leq 127$, binary-encode it in the 7 available bits and set $c = 1$
- Else encode G 's lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm
- At the end set the continuation bit of the last byte to 1 ($c = 1$) – and for the other bytes $c = 0$.

Example VB encoding. We start with the binary representation:

Dec: 1 5 1000 7 100 299999

Bin: 1 101 1111101000 111 1100100 1001001001111011111

The binary digits alone are not uniquely decodable:

111 comes from 7 or from three 1's ?

Variable byte (7+1) encoding

We use 7 bits for the binary digits and 1 continuation bit

1 10000001

5 10000101

1000 01101000 10000111

299999 -> 01011111 001001111 10001001

The continuation bit 1 means the current is the last byte of the encoding

Example

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Postings stored as the byte concatenation

000001101011100010000101000011010000110010110001

Key property: VB-encoded postings are uniquely prefix-decodable.

For a small gap (5), VB uses a whole byte.

Other variable unit codes

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles).
- Variable byte alignment wastes space if you have many small gaps – nibbles do better in such cases.
- Variable byte codes:
 - Used by many commercial/research systems
 - Good low-tech blend of variable-length coding and sensitivity to computer memory alignment matches (vs. bit-level codes, which we look at next).
- There is also recent work on word-aligned codes that pack a variable number of gaps into one word

- This doesn't look promising, but....

Gamma codes

- We can compress better with bit-level codes
 - The Gamma code is the best known of these.
- Represent a gap G as a pair *length* and *offset*
- *offset* is G in binary, with the leading bit cut off
 - For example $13 \rightarrow 1101 \rightarrow 101$
- *length* is the length of *offset*
 - For 13 (offset 101), this is 3.
- We encode *length* with *unary code*: 1110.
- Gamma code of 13 is the concatenation of *length* and *offset*: 1110101

Gamma code examples

number	length	offset	γ -code
0			none
1	0		0
2	10	0	10,0
3	10	1	10,1
4	110	00	110,00
9	1110	001	1110,001
13	1110	101	1110,101
24	11110	1000	11110,1000
511	111111110	11111111	111111110,11111111
1025	11111111110	0000000001	11111111110,0000000001

Gamma code properties

- G is encoded using $2 \lfloor \log G \rfloor + 1$ bits
 - Length of offset is $\lfloor \log G \rfloor$ bits
 - Length of length is $\lfloor \log G \rfloor + 1$ bits
- All gamma codes have an odd number of bits
- Almost within a factor of 2 of best possible, $\log_2 G$
- Gamma code is uniquely prefix-decodable, like VB
- Gamma code can be used for any distribution
- Gamma code is parameter-free

Gamma seldom used in practice

- Machines have word boundaries – 8, 16, 32, 64 bits
 - Operations that cross word boundaries are slower
- Compressing and manipulating at the granularity of bits can be slow
- Variable byte encoding is aligned and thus potentially more efficient
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost

RCV1 compression

Data structure	Size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
with blocking, $k = 4$	7.1
with blocking & front coding	5.9
collection (text, xml markup etc)	3,600.0
collection (text)	960.0
Term-doc incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ -encoded	101.0