Index Compression

Lecture 4

Why compression (in general)?

- Use less disk space
  - Save a little money; give users more space
- Keep more stuff in memory
  - Increases speed
- Increase speed of data transfer from disk to memory
  - [read compressed data | decompress] is faster than [read uncompressed data]

- Premise: Decompression algorithms are fast
  - True of the decompression algorithms we use

Postings compression

- 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.

Compression Example

- *Fixed Length* encoding –clear how to decode since number of bits is fixed

01010011101011001101

01010, 01110, 10110, 01101

10, 14, 22, 13

Number	Fixed Length Code (5 bit)
1	00001
2	00010
15	01111
17	10001
25	11001
31	11111

Fixed Length Encoding

- How many total bits are required for encoding for 10 million numbers using 20 bits for each number?
- 200 million bits = $2*10^8$ bits
- What if most numbers are small ?
- We are wasting many bits by storing small numbers in 20 bits.

Variable Length Encoding

- Decode 10110011
- 1011, 0,0,11 = 11, 0,0,3
- 10, 1100, 11 = 2, 12, 3
- 101,100, 11 = 5, 4, 3
- 10,11,0,0,11 = 2, 3, 0, 0, 3
-

Number	Variable Length code
1	1
2	10
10	1010
16	10000

Compression Example

Decode 0101011101100

- use unambiguous code:

Prefix free code

0, 1, 0, 3, 0, 2, 0

- which gives: $0\ 101\ 0\ 111\ 0\ 110\ 0$

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 $\log_2 1M \approx 20$ bits.
- A term like ***the*** occurs in virtually every doc, so 20 bits/posting ≈ 2MB is too expensive.

Gap encoding of postings file entries

- 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.
  - Especially for common words

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								

Index compression

- Observation of posting files
  - Instead of storing docID in posting, we store gap between docIDs, since they are ordered
  - Zipf's law again:
    - The more frequent a word is, the smaller the gaps are
    - The less frequent a word is, the shorter the posting list is
  - Heavily biased distribution gives us great opportunity of compression!

Information theory: entropy measures compression difficulty.

Delta Encoding

- Word count data is good candidate for compression
  – many small numbers and few larger numbers
  – encode small numbers with small codes
- Document numbers are less predictable
  – but differences between numbers in an ordered list are smaller and more predictable
- *Delta encoding*:
  – encoding differences between document numbers (*d-gaps*)

Delta Encoding

– Inverted list (without counts)

$$1, 5, 9, 18, 23, 24, 30, 44, 45, 48$$

– Differences between adjacent numbers

$$1, 4, 4, 9, 5, 1, 6, 14, 1, 3$$

– Differences for a high-frequency word are easier to compress, e.g.,

$$1, 1, 2, 1, 5, 1, 4, 1, 1, 3, \ldots$$

– Differences for a low-frequency word are large, e.g.,

$$109, 3766, 453, 1867, 992, \ldots$$

Practice Question

- Delta encode following numbers:
- 40, 45, 405, 411, 416
- 40, 5, 360, 6, 5

- Decode following numbers encoded using delta encoding
- 20, 10, 30, 4, 8
- 20, 30, 60, 64, 72

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 ~$\log_2 G$ bits/gap entry.
- <u>Key challenge</u>: 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

Unary Codes

- Breaks between encoded numbers can occur after any bit position
- *Unary* code
  - Encode *k* by *k* 1s followed by 0
  - 0 at end makes code unambiguous

Number	Code
0	0
1	10
2	110
3	1110
4	11110
5	111110

1110111010110

1110, 1110, 10, 110

3, 3, 1, 2

Unary Codes

- Unary is very efficient for small numbers such as 0 and 1, but quickly becomes very expensive
  - 1023 can be represented in 10 binary bits, but requires 1024 bits in unary

Binary Codes

- Variable Length Binary Codes:
  - Variable length Binary is more efficient for large numbers, but it may be ambiguous
  - E.g. 2 encoded as 10 and 5 encoded as 101
  - 10101, how to decode, we don’t know the word bound
- Fixed Length Binary Codes:
  - For example use 15 bits to encode each number
  - Cannot encode large numbers that required more than 15 bits
  - Too much space is wasted for encoding small numbers

Elias-γ Code (Elias Gamma Code)

- Encode number in minimum bits in binary

- 12,
- 1100 ,   4 bits
- Encode length of the code in unary in begining
- Code of 12 = 111101100

- 23
- 10111 ,   5 bits
- Code of 23 = 1111110111

Elias-γ Code (Elias Gamma Code)

- Decode 1110 101 110 10

- 1110101 , 11010
- 5 , 2

Elias-γ Code (Elias Gamma Code)

- Encode number in minimum bits in binary

- 12,
- 1100 , 4 bits
- 100, leave the leftmost 1 bit, length 3 bits
- Code of 12 = 1110100

- 23
- 10111 , 5 bits
- 0111, 4 bits, leave the leftmost 1 bit
- Code of 23 = 111100111

Elias-γ Code (Elias Gamma Code)

- Decode 111001111000
- 1110 011, 110 00
- 011 was actually 1011 = 11
- 00 was actually 100 = 4
- 11, 4

Elias-γ Code (Elias Gamma Code)

- To encode a number *k*, compute
  - $k_d = \lfloor \log_2 k \rfloor$
  - $k_r = k - 2^{\lfloor \log_2 k \rfloor}$
  - – *Since the leftmost bit is always 1 in binary code so we do not encode it*
  - – *The remaining number becomes $k_r = k - 2^{lgk}$*
  - – *We use Unary code for k_d and binary code for k_r*

Elias-γ Code (Elias Gamma Code)

- To encode a number *k*, compute
  - $k_d = \lfloor \log_2 k \rfloor$
  - $k_r = k - 2^{\lfloor \log_2 k \rfloor}$

Unary code for k_d and binary code for k_r

Number (k)	k_d	k_r	Code
1	0	0	0
2	1	0	10 0
3	1	1	10 1
6	2	2	110 10
15	3	7	1110 111
16	4	0	11110 0000
255	7	127	11111110 1111111
1023	9	511	1111111110 111111111

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 too slow

- All modern practice is to use byte or word aligned codes
  - Variable byte encoding is a faster, conceptually simpler compression scheme, with decent compression

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 <u>continuation</u> 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 $c = 0$.

Variable Byte Code

Consider the vByte representation of the postings list L = ⟨80, 400, 431, 686⟩:

$\Delta(L)$ = ⟨80, 320, 31, 255⟩

vByte(L) = $\underline{1}1010000$ $\underline{0}1000000$ $\underline{1}0000010$ $\underline{1}0011111$ $\underline{0}1111111$ $\underline{1}0000001$

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.

RCV1 compression

Data structure	Size in MB
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

Index and dictionary compression for Reuters-RCV1.
(Manning et al. Introduction to Information Retrieval)

Group Variable Integer code

- Used by Google around turn of millennium….
  - Jeff Dean, keynote at WSDM 2009
  - Encodes 4 integers in blocks of size 5–17 bytes
- First byte: four 2-bit binary length fields
- | L_1 | L_2 | L_3 | L_4 | , $L_j \in \{1,2,3,4\}$
- Then, L1+L2+L3+L4 bytes (between 4–16) hold 4 numbers
  - Each number can use 8/16/24/32 bits. Max gap length ~4 billion
- It was suggested that this was about twice as fast as VB encoding
  - Decoding gaps is much simpler – no bit masking
  - First byte can be decoded with lookup table or switch

Group Variable Integer code

Consider the vByte representation of the postings list L = ⟨80, 400, 431, 686⟩:

Δ(L) =	⟨80, 320, 31, 255⟩
vByte(L) =	11010000 01000000 10000010 10011111 01111111 10000001

For the same postings list as before, the Group VarInt representation is:

GroupVarInt(L) = 00010000 01010000 01000000 00000001 00011111 11111111

Compression Techniques Effectiveness
(All Data from Gov2 Collection)

		Decoding (ns per position)	Cumulative Overhead (decoding + disk I/O)
Gamma		12.81	32.11 ns
vByte		4.34	20.82 ns
Group VarInt		1.9	19.85 ns

Chapter 6, Information Retrieval: Implementing and Evaluating Search Engines, by S. Büttcher, C. Clarke, and G. Cormack.