Chapter IR:IV

IV. Indexes

- Inverted Indexes
- Query Processing
- Index Construction
- Index Compression

Compression

Size Issues

- Inverted lists are very large
  - E.g., 25–50% of collection for TREC collections using Indri search engine
  - Much higher if n-grams are indexed
- Compression of indexes saves disk and/or memory space
  - Typically have to decompress lists to use them
  - Best compression techniques have good compression ratios and are easy to decompress
  - Allows data to move up the memory hierarchy
  - Reduces seek time on disk
- Disadvantage: Decompression time
- Here: Lossless compression – no information lost
  - Lossy compression for images, audio, video with very high compression ratios

Compression

Savings

- Processor can process p inverted list postings per second
- Memory system can supply processor with m postings per second
- Number of postings processed each second: min$\{m, p\}$.
  - If $p > m$, the processor will spend some of its time waiting for postings to arrive from memory
  - If $m > p$, the memory system will sometimes be idle
- Compression ratio r, decompression factor d
  - Memory supplies rm postings per second
  - Processor processes dp postings per second
  - Number of postings processed each second: min$\{rm, dp\}$
- No compression: $r = d = 1$
- Reasonable: $r > 1$ and $d < 1$
  - Compression useful only if $p > m$
  - Ideal: $rm = dp$

Compression

Basic Idea

- Common data elements use short codes while uncommon data elements use longer codes
- Inverted lists are lists of numbers
- Example: Coding numbers
  - Number sequence: 0, 1, 0, 2, 0, 3, 0
  - Possible encoding (2 bits): 00 01 00 10 00 11 00
  - Encode 0 using a single 0: 0 01 0 10 0 11 0
  - Only 10 bits, but looks like: 0 01 01 0 0 11 0
  - Which encodes: 0, 1, 1, 0, 0, 2, 0
    - Ooops!
  - Better: Unambiguous code
    - $0 \rightarrow 0, \quad 1 \rightarrow 101, \quad 2 \rightarrow 110, \quad 3 \rightarrow 111$
    - Yields 0 101 0 111 0 110 0 (13 bits)
    - 2-bit encoding was also unambiguous (14 bits)

Compression

Delta Encoding

- Entropy measures predictability of input
- Word count data is good candidate for compression
  - Most words are rare
  - Many small numbers and few larger numbers
  - Encode small numbers with small codes
- Document numbers are less predictable
  - Longer documents occur more often in index
  - But not a lot of entropy in the distribution; no large effect
- Idea: Differences between numbers in an ordered list are smaller and more predictable
- Delta encoding: Encode differences between document numbers (d-gaps)

Compression

Delta Encoding

- Inverted list (without counts etc.)
  - 1, 5, 9, 18, 23, 24, 30, 44, 45, 48
- Differences between adjacent numbers (d-gaps)
  - 1, 4, 4, 9, 5, 1, 6, 14, 1, 3
  - Advantage: Ordered list of (large) numbers turns into list of small numbers
- Differences for a high-frequency word are easier to compress
  - 1, 1, 2, 1, 5, 1, 4, 1, 1, 3, ...
- Differences for a low-frequency word are large
  - 109, 3766, 453, 1867, 992, ...
  - Bad: Large numbers
  - Good: List is short

Compression

Bit-Aligned Codes

- Breaks between encoded numbers can occur after any bit position
  - Byte-aligned are more favorable to certain operating systems
- Goal: Small numbers receive small code values
- Unary code
  - Encode k by k 1s followed by 0
  - 0 at end makes code unambiguous
  - $0 \to 0, \quad 1 \to 10, \quad 2 \to 110, \quad 3 \to 1110, \ldots$
- Others: Elias-γ and Elias-δ

Compression

Unary and Binary Codes

- Unary is more 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 is more efficient for large numbers, but it may be ambiguous
  - Not so useful on its own for compression

Compression

Elias-γ Code

- To encode a number k, compute

  $$k_d = \lfloor \log_2 k \rfloor \qquad \text{and} \qquad k_r = k - 2^{\lfloor \log_2 k \rfloor}$$

  - k_d is number of binary digits
  - k_r is k after removing the leftmost 1 of its binary encoding ($k > 0$)
- Idea: Encode k_d as unary and k_r as binary (in k_d binary digits)
  - Unary part tells us how many binary digits to expect

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

Compression

Elias-δ Code

- Elias-γ code uses no more bits than unary, many fewer for $k > 2$
  - 1023 takes 19 bits instead of 1024 bits using unary
- In general, takes $2\lfloor \log_2 k \rfloor + 1$ bits
  - $\lfloor \log_2 k \rfloor + 1$ for unary part
  - $\lfloor \log_2 k \rfloor$ for binary part
- To improve coding of large numbers, use Elias-δ code
  - Instead of encoding k_d in unary, we encode $k_d + 1$ using Elias-γ
  - Takes approximately $2 \log_2 \log_2 k + \log_2 k$ bits

Compression

Elias-δ Code

- Split k_d into: $k_{dd} = \lfloor \log_2(k_d + 1) \rfloor$ and $k_{dr} = k_d - 2^{\lfloor \log_2(k_d+1) \rfloor}$
  - Encode k_{dd} in unary, k_{dr} in binary, and k_r in binary

Number (k)	k_d	k_r	k_{dd}	k_{dr}	Code
1	0	0	0	0	0
2	1	0	1	0	10 0 0
3	1	1	1	0	10 0 1
6	2	2	1	1	10 1 10
15	3	7	2	0	110 00 111
16	4	0	2	1	110 01 0000
255	7	127	3	0	1110 000 1111111
1023	9	511	3	2	1110 010 111111111

- Sacrifices efficiency for low numbers for smaller encodings of large numbers
  - Numbers between 16 and 32 require same space as Elias-γ
  - Numbers larger than 32 require less space

Compression

Byte-Aligned Codes

- Variable-length bit encodings can be a problem on processors that process bytes
- v-byte is a popular byte-aligned code
  - Similar to Unicode UTF-8
- Short codes for small numbers
  - But shortest v-byte code is 1 byte
  - 8 times longer than Elias-γ for number 1
- Numbers are 1 to 4 bytes, with high bit set to 1 in the last byte, 0 otherwise
- Byte-aligned codes compress and decompress faster

Compression

V-Byte Encoding

k	Number of bytes
$k < 2^7$	1
$2^7 \leq k < 2^{14}$	2
$2^{14} \leq k < 2^{21}$	3
$2^{21} \leq k < 2^{28}$	4

k	Binary Code	Hexadecimal
1	1 0000001	81
6	1 0100110	86
127	1 1111111	FF
128	0 0000001 1 0000000	01 80
130	0 0000001 1 0000010	01 82
20,000	0 0000001 0 0011100 1 0100000	01 1C A0

Compression

Example

- Original inverted list with positions (docID, position)
  - (1001,1) (1001,7) (1002,6) (1002,17) (1002,197) (1003,1)
- Group positions for each document (docID, count, [positions]):
  - (1001,2,[1,7]) (1002,3,[6,17,197]) (1003,1,[1])
  - Count makes list decipherable even without brackets
    - 1001,2,1,7,1002,3,6,17,197,1003,1,1
- Delta encode document numbers and positions to make numbers smaller:
  - (1,2,[1,6]) (1,3,[6,11,180]) (1,1,[1])
  - Count cannot be delta-encoded with previous techniques (some extra work)
- Compress 1,2,1,6,1,3,6,11,180,1,1,1 using v-byte:
  - 81 82 81 86 81 82 86 8B 01 B4 81 81 81
  - 13 Bytes for entire list