Chapter IR:II

II. Indexing

- Indexing Basics
- Inverted Index
- Query Processing I
- Query Processing II
- Index Construction
- Index Compression
- Size Estimation

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Size Issues

Inverted lists can become very large.

- Rule of thumb: 25–50% of document collection.
- 2–4 times higher if n-grams are indexed.

Compression of indexes saves disk and/or memory space.

- Best techniques have good compression ratios, easy to decompress.
- Reduces seek time on disk.
- Disadvantage: Decompression time.

We need lossless compression ➜ no information lost

- Lossy compression for images, audio, video with very high compression ratios

As we iterate the posting list, read a stream of bits to decode postings.

- Postings must be decoded while reading.

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Basic Idea

Common elements use short codes, uncommon elements use long codes.

- Posting lists are just lists of numbers.

Naïve number coding:

- 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 ➜ 0 is common element
- Only 10 bits, but looks like: 0 01 01 0 0 11 0
- Which encodes: 0, 1, 1, 0, 0, 3, 0
  - Oops!

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Basic Idea

Common elements use short codes, uncommon elements use long codes.

- Posting lists are just lists of numbers.

Naïve number coding:

- 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 ➜ 0 is common element
- Only 10 bits, but looks like: 0 01 01 0 0 11 0
- Which encodes: 0, 1, 1, 0, 0, 3, 0
  - Oops!

Unambiguous coding:

- 0 → 0, 1 → 101, 2 → 110, 3 → 111 ➜ add a 1 before each number
- Yields 0 101 0 110 0 111 0 (13 bits)
- 2-bit encoding was also unambiguous (14 bits)

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Unambiguous codes

Goal: Small numbers receive small code values ➜ Unary code.

- Encode k by k 1s followed by 0 (0 at end makes code unambiguous).
- $0 \rightarrow 0, \quad 1 \rightarrow 10, \quad 2 \rightarrow 110, \quad 3 \rightarrow 1110, \ldots$

Unary: efficient for small numbers such as 0 and 1, but quickly becomes expensive.

- 1023 can be represented in 10 binary bits, but requires 1024 bits in unary.

Binary: efficient for large numbers, but it may be ambiguous (not byte aligned).

- Not so useful on its own for compression.

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Elias-γ Code

Let's use advantages from unary and binary encoding schemes.

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

- k_d is least amount of binary digits needed (highest power of 2).
- k_r is k after removing the leftmost 1 of its binary encoding ($k > 0$).

Encode: k_d as unary (followed by 0) and k_r as binary (in k_d binary digits).

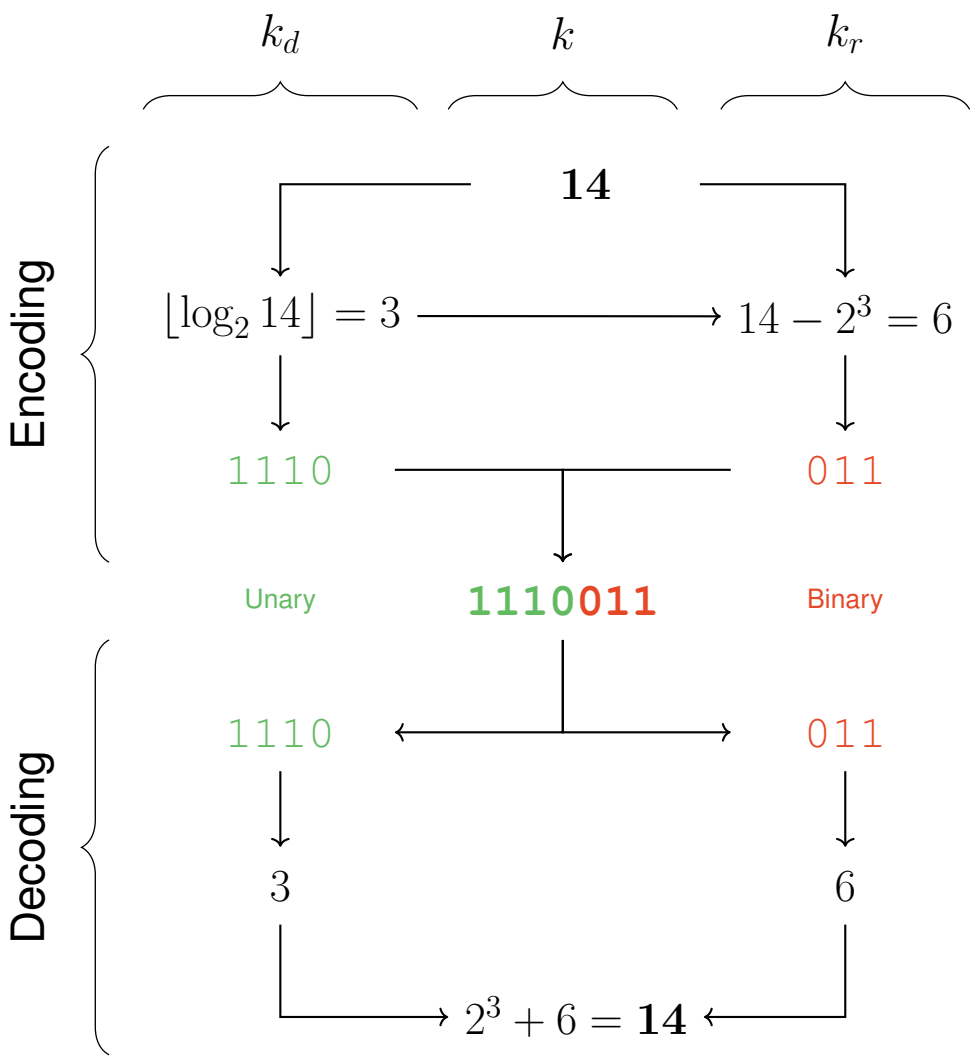
Number (k)	k_d	k_r	Code
1	0	0	0
2	1	0	10 0
6	2	2	110 10
15	3	7	1110 111
16	4	0	11110 0000
255	7	127	11111110 1111111
1023	9	511	1111111110 111111111

Decode: Let N be k_d without the final 0, then $2^N + k_r$.

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Elias-γ Code



©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Elias-γ Code

Elias-γ code never uses more bits than unary, many fewer for $k > 2$

- 1023 takes just 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

In binary, can encode k in $\lfloor \log_2 k \rfloor$ bits.

- Elias-γ needs twice as much as binary to make it unambiguous.

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Elias-γ Code

Elias-γ code never uses more bits than unary, many fewer for $k > 2$

- 1023 takes just 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

In binary, can encode k in $\lfloor \log_2 k \rfloor$ bits.

- Elias-γ needs twice as much as binary to make it unambiguous.

We have an unambiguous code.

What can we encode such that we have few large numbers?

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Delta Encoding

What to compress? ➜ Need to find distributions with few large numbers.

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Delta Encoding

What to compress? ➜ Need to find distributions with few large numbers.

Document identifiers.

- Longer documents occur more often in an index.

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Delta Encoding

What to compress? ➜ Need to find distributions with few large numbers.

Document identifiers.

- Longer documents occur more often in an index.

Differences between document identifiers in a posting list.

- Document identifiers grow but distances between are on average the same.
- Differences in document identifiers are mostly small numbers.
- Delta encoding: Encode differences between document numbers (d-gaps)

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Delta Encoding

Posting list of document ids.

- 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

Ordered list of (large) numbers turns into ordered list of small numbers.

- We can still do better than Elias-γ when we have large gaps.

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 $2 \log_2 k + \log_2 k$ bits (as opposed to $2\lfloor \log_2 k \rfloor + 1$).

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

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

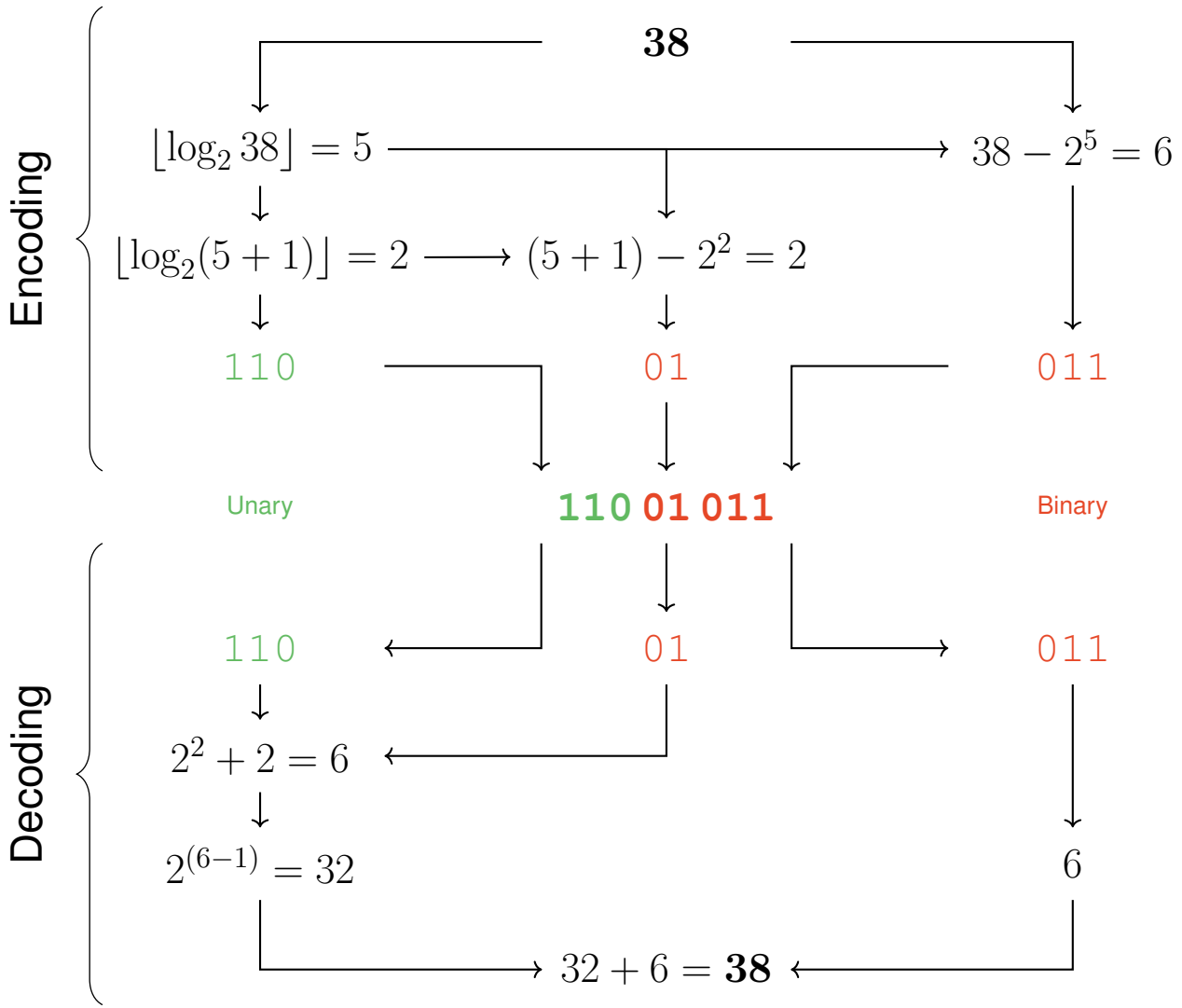
Number (k)	k_d	k_r	k_{dd}	k_{dr}	Code
1	0	0	0	0	0
2	1	0	1	0	10 0 0
6	2	2	1	1	10 1 10
15	3	7	2	0	110 00 111
16	4	0	2	1	110 01 0000
255	7	127	3	0	1110 000 1111111
1023	9	511	3	2	1110 010 111111111

- Decode: Count the L ones until first zero (k_{dd}); read another L bits after the zero (k_{dr}); decode all using Elias-γ decoding (N). Decode remaining bits as binary (k_r). Final decoded value is $(N - 1) + k_r$.
- Produces longer encodings of small numbers than Elias-γ (<16, same space between 16 and 32)
- Produces shorter encodings of large numbers than Elias-γ (>32)

©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022

Compression

Elias-δ Code



©GIENAPP/HAGEN/POTTHAST/SCELLS/STEIN 2022