Index and document compression

IN4325 – Information Retrieval

Last time

High-level view on …

- (Basic, positional) inverted index
- Biword index
- Hashes versus search trees for vocabulary lookup
- Index structures for wildcard queries (permuterm index, etc.)

Today

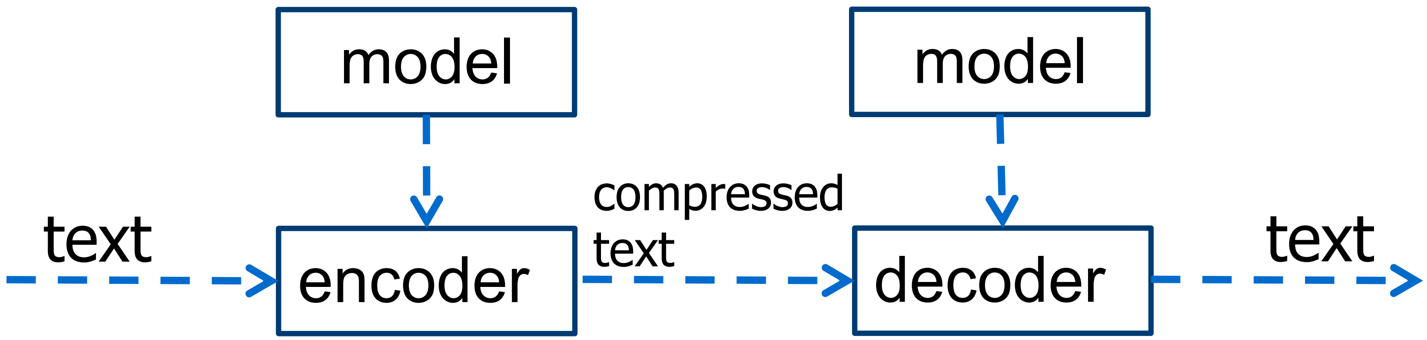
- Efficiency needed in the three stages
    - Construction of the index/indices
        - Web search engines need to scale up to billions of documents
    - Storage of indices
    - Storage of documents

- Inverted file creation
- Dictionary compression techniques
- Inverted file compression techniques
- Document compression techniques

} index compression

Text compression

Lossless!



Inverted file creation

$$< t; df_t; (d_1, f_{t1}), (d_2, f_{t2}), \ldots, (d_{f_t}, f_{df_t}) >, d_i < d_j \ \forall i < j$$

$$< t; df_t; (d_1, f_{t1}; (pos_1, pos_2, \ldots, pos_{f_{t1}})), \ldots >, d_i < d_j \ \forall i < j$$

- How can the dictionary and posting lists be created from a corpus of documents?
  - Posting lists file (on disk) is orders of magnitude larger than the dictionary file (in memory for fast access)

- Scalability challenge
  - Millions of words of documents, billions of term occurrences
  - Severely memory limited environments (mobile devices)

Hardware constraints [10]

- Major constraints due to hardware
  - Disks maximize input/output throughput if contiguously stored data is accessed
  - Memory access is faster than disk access
  - Operating systems read/write blocks of fixed size from/to disk
  - reading compressed data from disk and decompressing it is faster than reading uncompressed data from disk

Index creation approaches

How to compare their running time analytically?

- Commonly employed computational model [4]
  - OS effects are ignored (caching, cost of memory allocation, etc.)
  - Values based on a Pentium III 700MHz machine (512MB memory)

Parameters		value
Main memory size (MB)	M	256
Average disk seek time (sec)	t_s	$9x10^{-3}$
Disk transfer time (sec/byte)	t_t	$5x10^{-8}$
Time to parse one term (sec)	t_p	$8x10^{-7}$
Time to lookup a term in the lexicon (sec)	t_l	$6x10^{-7}$
Time to compare and swap two 12 byte records (sec)	t_w	$1x10^{-7}$

Index creation approaches

How to compare their running time analytically?

- Commonly employed computational model [4]
  - TREC corpus (20GB Web documents)

Parameters		value
Size (Mb)	S	*20,480*
Distinct terms	n	*6,837,589*
Term occurrences ($x10^6$)	C	*1,261.5*
Documents	N	*3,560,951*
Postings $(d,f_{d,t})$ ($x10^6$)	t_l	$6x10^{-7}$
Avg. number of terms / document	C_{avg}	*354*
Avg. number of index terms /document	W_{avg}	*153*
%words occuring once only	H	*44*
Size of compressed doc-level inverted file (MB)	I	*697*
Size of compressed word-level inverted file (MB)	I_w	*2,605*

Inverted file creation

Simple in-memory inversion

Relying on the OS and virtual memory is too slow since list access (eq. matrix rows) will be in random order

① First pass over the collection to determine the number of unique terms (vocabulary) and the number of documents to be indexed

② Allocate the matrix and second pass over the collection to fill the matrix

③ Traverse the matrix row by row and write posting lists to file

- Prohibitively expensive [4]
  - Small corpus ➔2 bytes per *df*: 4.4MB corpus yields 800MB matrix
  - Larger corpus➔4 bytes per *df*: 20G corpus yields 89TB matrix

- Alternative: list-based in-memory inversion (one list per term)
  - Each node represents $(d,f_{d,t})$ and requires 12 bytes (posting+pointer)
  - The 20G corpus requires 6G main memory [4]

Disk-based inversion

Candela and Harman, 1990 [5]

Predicted indexing time for the 20GB corpus: 28.3 days.

- Requires a single pass
- Writes postings to temporary file, lexicon resides in memory

① Initialize lexicon structure in memory (keeps track of the last posting file adress p of term t in the temp. file)

② Traverse the corpus (sequential disk access)

  ① Fr each posting $(t,d,f_{d,t})$, query the lexicon for t and retrieve p

  ② Append temporary file: add $(t,d,f_{d,t},p)$ & update lexicon (p')

③ Post-process the temporary file to create an inverted file

  ① Allocate a new file on disk

  ② For each term (lexicographic order), traverse the posting list in reverse order, compress the postings and write to inverted file (**random disk access**)

Disk-based inversion

Candela and Harman, 1990 [5]

Predicted indexing time for the 20GB corpus: 28.3 days.

- Requires a single pass
- Writes postings to temporary file, lexicon resides in memory

Predicted inversion time (in seconds):

$$T = St_t + C(t_p + t_l) + 10Pt_t +$$ read, parse, lookup lexicon, write postings

$$Pt_s / v + 10Pt_t +$$ traverse lists

$$I(t_c + t_t)$$ compress, write out to inverted file

① Initial
last p
② Trave
   ① Fr
   ② Ap
③ Post-
   ① Allocate a new file on disk
   ② For each term (lexicographic order), traverse the posting list in reverse order, compress the postings and write to inverted file (**random disk access**)

Sort-based inversion

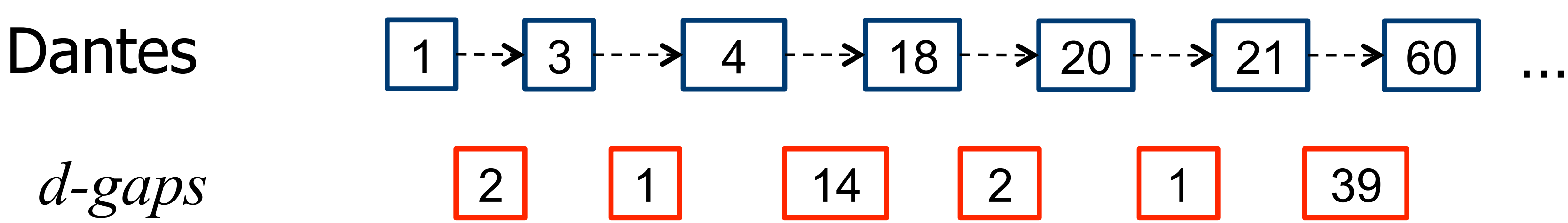
1. Create empty dictionary structure S and empty temporary file on disk
2. For each document d in the collection
   1. Process d and then for each parsed index term t
      1. If t is not in S, insert it (in memory)
      2. Write $<t,d,f_{d,t}>$ to temporary file
3. Sort: assume k records can be held in memory; read file in blocks of k records
   1. Read *k* records from temporary file
   2. Sort into non-descending t order and then d order (e.g. Quicksort)
   3. Write sorted-k-run back to temporary file
4. Pairwise merge runs in the temporary file until entire file is sorted (from R initial runs $\lceil \log_2 R \rceil$ merges are required)
5. Output inverted file: for each term t
   1. Start a new inverted file entry
   2. Read all $<t,d,f_{d,t}>$ from temporary file
   3. Append this inverted list to the inverted file

d-gaps

Compressing the posting lists

- We store *positive integers* (document identifiers, term pos.)
- If upper bound for x is known, x can be encoded in $\lceil \log_2 X \rceil$ bits
  - 32-bit unsigned integers: $0 \leq x < 2^{32}$

- Inverted lists can also be considered as a sequence of run length or **document gaps** between document numbers [7]



d-gaps

Compressing the posting lists: unary code

The binary code assumes a uniform probability distribution of gaps.

- **d-gaps** are
  - Small for frequent terms
  - Large for infrequent terms

- Basic idea: encode small value integers with short codes

- **Unary code** (global method): an integer x (gap) is encoded as $(x\text{-}1)$ one bits followed by a single zero bit
  - Assumed probability distribution of gaps: $P(x) = 2^{-x}$

1 ➔0
2 ➔10
22➔1111111111111111111110

d-gaps

Compressing the posting lists: Elias's γ code (1975) [8]

- Elias's γ code:

$$x \text{ as unary code for } 1+\lfloor \log_2 x \rfloor$$

$$\text{followed by a code of } \lfloor \log_2 x \rfloor \text{ bits coding } x - 2^{\lfloor \log_2 x \rfloor} \text{ in binary}$$

#bits to encode x

codes x in that many bits

- Assumed probability distribution: $P(x) = \frac{1}{2x^2}$

- Example 1

$$\text{value } x = 5$$

$$\lfloor \log_2 x \rfloor = 2$$

$$\text{coded in unary}: 3 = 1+2 \text{ (code } 110)$$

$$\text{followed by } 1 = 5-4 \text{ as a two-bit binary (code } 01)$$

$$\text{codeword}: 11001$$

d-gaps

Compressing the posting lists: Elias's γ code (1975) [8]

- Elias's γ code example 2

$$value\ x = 8$$
$$\lfloor \log_2 x \rfloor = 3$$
$$coded\ in\ unary: 4 = 1 + 3\ (code\ 1110)$$
$$followed\ by\ 0 = 8 - 8\ as\ a\ three-bit\ binary\ (code\ 000)$$
$$codeword: 1110000$$

- Unambiguous decoding
  ① Extract unary code c_u
  ② Treat the next c_u-1 bits as binary code to get c_b

$$x = 2^{c_u - 1} + c_b$$

d-gaps

Compressing the posting lists: Elias's δ code (1975) [8]

- Elias's δ code:

$$x \text{ as } \gamma \text{ code for } 1+\lfloor \log_2 x \rfloor$$

$$\text{followed by a code of } \left[\log_2 x\right] \text{ bits coding } x - 2^{\lfloor \log_2 x \rfloor} \text{ in binary}$$

- Example

$$\text{value } x = 5$$

$$\lfloor \log_2 x \rfloor = 2$$

$$\text{coded in } \gamma - \text{code}: 3 = 1 + 2 \ (\text{code } 101)$$

$$\text{followed by } 1 = 5 - 4 \text{ as a two} - \text{bit binary } (\text{code } 01)$$

$$\text{codeword}: 10101$$

- Number of bits required to encode x: $1 + 2\lfloor \log_2 \log_2 2x \rfloor + \lfloor \log_2 x \rfloor$

d-gaps

Compressing the posting lists: Golomb code (1966) [9]

- Golomb code (local method):
  - Different inverted lists can be coded with different codes (change in parameter b: dependent on corpus term frequency)
  - Obtains better compression than non-parameterized Elias's codes

$$parameter\ b = 0.69(N / f_t)$$
$$x\ as\ (q+1)\ unary, where\ q = \lfloor (x-1)/b \rfloor$$
$$followed\ by\ r = (x-1) - q \times b\ coded\ in\ binary$$
$$(requires\ \lfloor \log_2 b \rfloor\ or\ \lceil \log_2 b \rceil\ bits)$$

Requires two passes to generate!

- Example

$$value\ x = 5, assume\ b = 3$$
$$q = \lfloor (5-1)/3 \rfloor = 1 + 1\ (code\ 10)$$
$$r = (5-1) - 1 \times 3 = 1\ (code\ 10)$$
$$codeword: 1010$$

Examples of encoded d-gaps

Gap x	Unary	Elias's γ	Elias's δ	Golomb b=3
1	0	0	0	00
2	10	100	1000	010
3	110	101	1001	011
4	1110	11000	10100	100
5	11110	11001	10101	1010
6	111110	11010	10110	1011
7	1111110	11011	10111	1100
8	11111110	1110000	11000000	11010
9	111111110	1110001	11000001	11011
10	1111111110	1110010	11000010	11100

d-gaps

Adding compression to positional posting lists

In practice compress [4]:

- d-gaps with Golomb codes
- $f_{d,t}$ and word-position gaps with Elias codes

- So far, we considered the document gaps
- In positional postings, we also have $f_{d,t}$ values
  - Often one, rarely large

method	d-gaps	$f_{d,t}$
Unary		1.71
binary	21.00	
Elias's γ	6.76	1.79
Elias's δ	6.45	2.01
Golomb	6.11	

Inverted file compression for a 2G TREC collection (2 million records, 1000 bytes each) [6].
Index contains 196 million pointers in total and requires 185M disk space.

Results in bits per pointer.

Sort-based inversion II

Moffat and Bell, 1995 [6]

lexicon

Index term	Term mum.
this	1
is	2
a	3
sample	4

$t,d,f_{d,t}$

$t',d,f_{d,t}$ (12 byte)

in-memory array

1,1,1
2,1,1
3,1,1
4,1,1
5,1,1
3,2,1
6,2,1
1,3,1
5,3,1
…

$t',d,f_{d,t}$

1,1,1
1,3,1
2,1,1
3,1,1
3,2,1
4,1,1
5,1,1
5,3,1
6,2,1
…

temp. file

parse documents

map terms to term number

accumulate postings

sort array and write compressed run (memory threshold reached)

accumulate runs on disk

Adapted from: [4]

Sort-based inversion II

Moffat and Bell, 1995 [6]

Predicted indexing time for the 20GB corpus: 105 minutes.

- Compress the temporary file ($<t,d,f_{d,t}>$ triples)
  - Elias's δ code for d-gaps (Golomb would require 2 passes again)
  - Elias's γ for $f_{d,t}$ components
  - Representation of the t component, e.g. unary
    - Remove the randomness in the unsorted temporary file by interleaving the processing of the text and the sorting the postings in memory
    - t-gaps are thus 0 or higher (triples are sorted by t!)
- K-way merge
  - Merging in one pass
  - in-situ replacement of the temporary file
- The lexicon needs to be kept in memory

Storing the inverted list by term ids is a problem for range queries.
Storage according to lexicographical order can be done in a second pass.

Efficient single pass index construction

Heinz and Sobel, 2004 [4]

- Previous approaches required the vocabulary to remain in main memory
  - Not feasible for very large coprora

Formally

- Zipf's law: collection term frequency decreases rapidly with rank

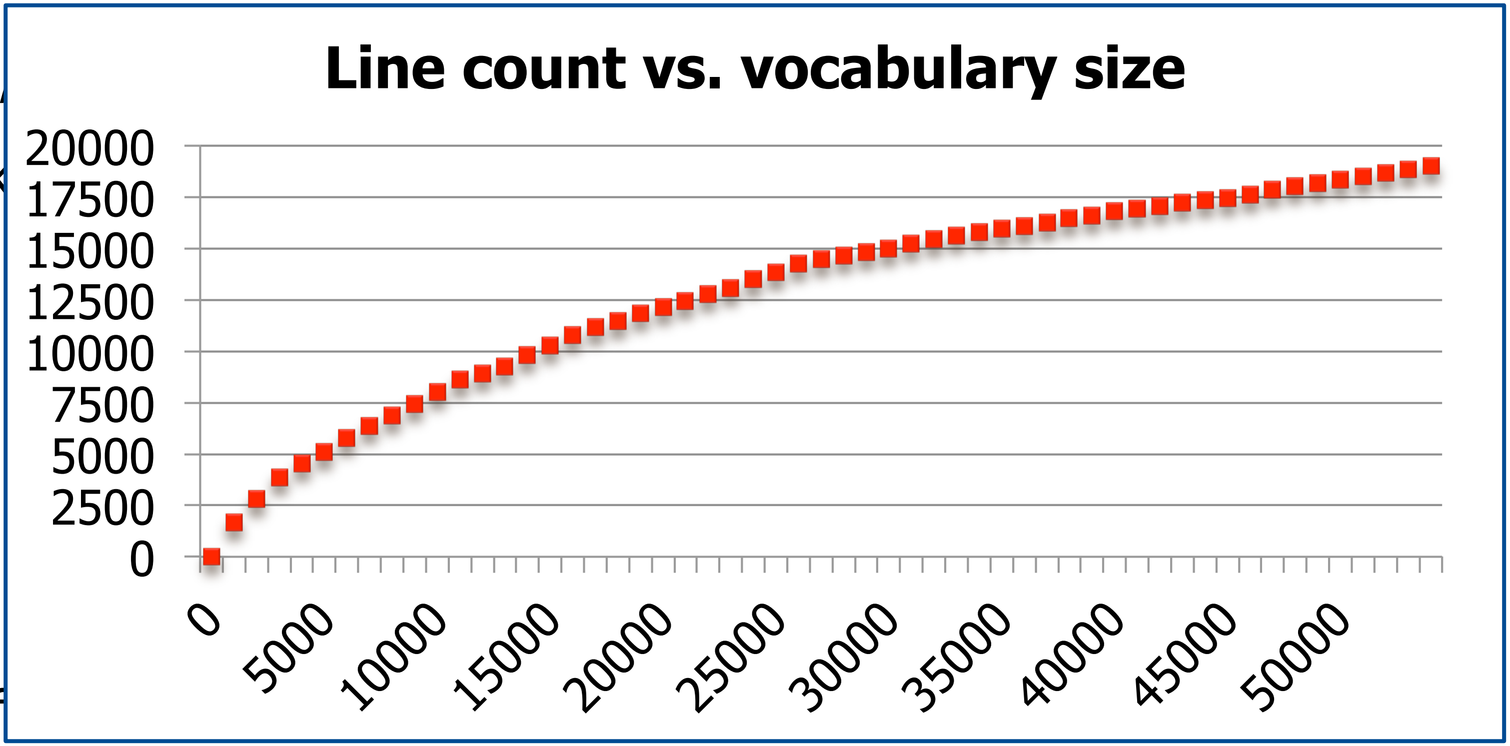
$$cf_i \propto \frac{1}{i}, where\ cf_i\ is\ the\ collection\ frequency$$
$$of\ the\ ith\ common\ term$$

- Heap's law: the vocabulary size V grows linearly with the size N of the corpus

$$V = kN^b, where\ N\ is\ \#tokens\ in\ the\ corpus$$
$$typically\ 30 \leq k \leq 100, b \approx 0.5$$

Formally

- Zipf'
rank



- Hea
N of

$$V = kN^b, \textit{where } N \textit{ is \# tokens in the corpus}$$

$$\textit{typically } 30 \le k \le 100, b \approx 0.5$$

Efficient single pass index construction

Heinz and Sobel, 2004 [4]

- Previous approaches required the vocabulary to remain in main memory
  - Not feasible for very large coprora
  - Heap's law: vocabulary increases "endlessly"
  - Zipf's law: many terms occur only once
    - i.e. inserted into the in-memory lexicon, but never accessed again

- Efficient single pass indexing offers a solution
  - Does not require all of the vocabulary to remain in main memory
  - Can operate within limited volumes of memory
  - Does not need large amounts of temporary disk space
  - Faster than previous approaches

Efficient single pass index construction II

Heinz and Sobel, 2004 [4]

- Based on the same ideas as sort-based inversion (in-memory construction of runs that are saved to disk and stored)
  - Design is crucial to achieve better results

- Main idea: assign each index term in the lexicon a dynamic in-memory vector that accumulates their corresponding postings in compressed form (Elias codes)
  - Last inserted document number needs to be known (kept as uncompressed integer)

Efficient single pass index construction II

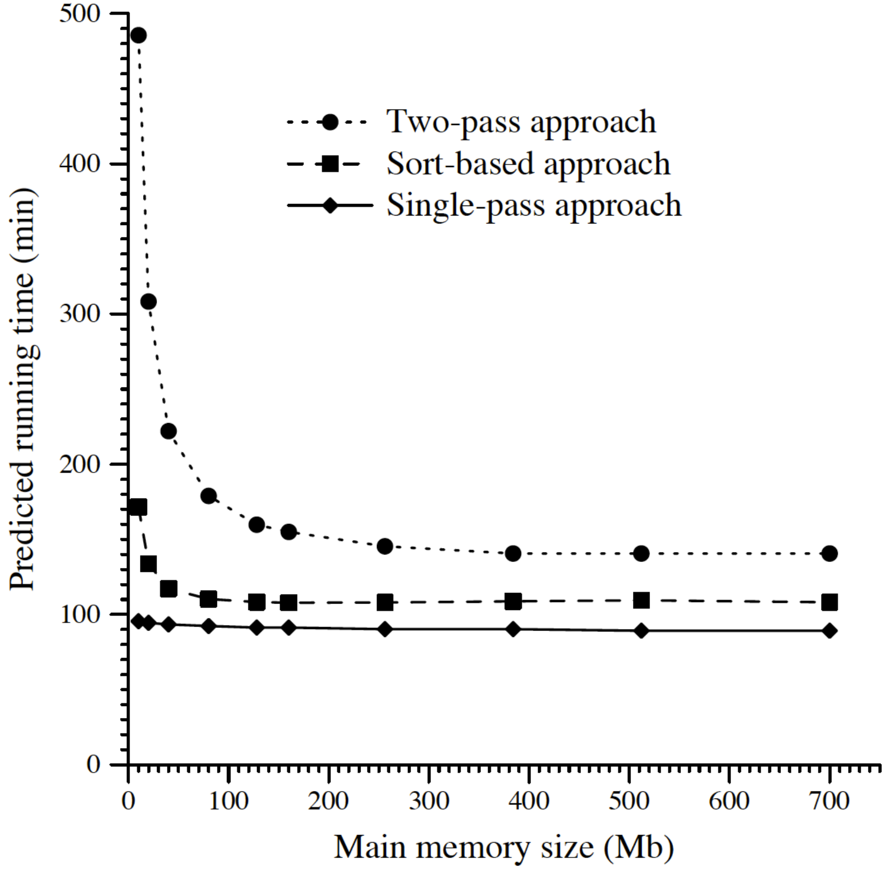
Heinz and Sobel, 2004 [4]

Predicted indexing time for the 20GB corpus: 91 minutes.

1. Allocate empty temporary file on disk
2. For each posting and as long as main memory is available, search the lexicon
   1. If not found, insert t into the lexicon, initialize bitvector
   2. Add posting to bitvector and compress on the fly
3. If main memory is used up, index terms and bitvectors are processed in lexicographic order
   1. Each index term is appended to the temporary file on disk (front-coding) together with the padded bitvector
   2. Lexicon is freed
4. Repeat steps 2&3 until all documents have been processed
5. Compressed runs are merged to obtain the final inverted file

Efficient single pass index construction II

Heinz and Sobel, 2004 [4]



Predicted running time in minutes over the 20G corpus.
Taken from [4].

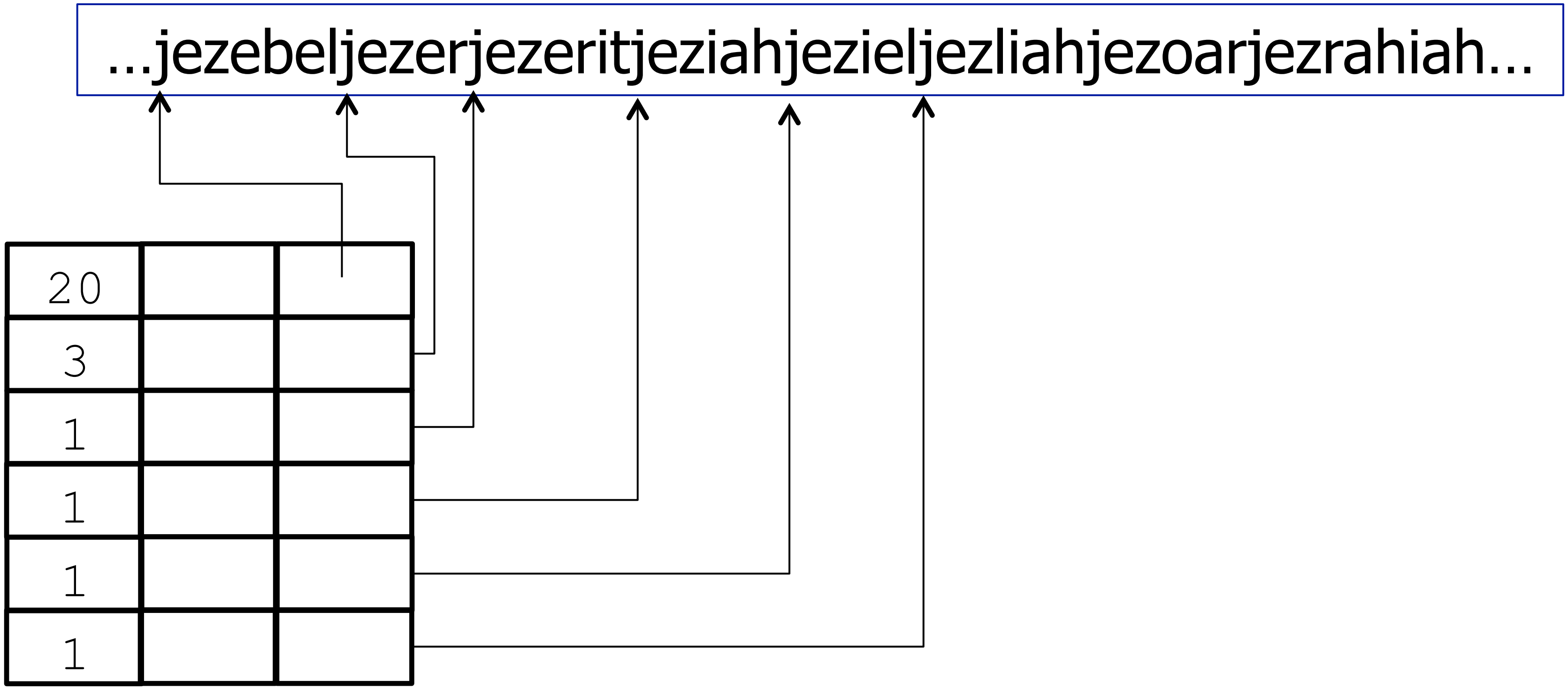
Recall: dictionary

term t	f_t	disk address of I_t
jezebel	20	→
jezer	3	→
jezerit	1	→
jeziah	1	→
jeziel	1	→
jezliah	1	→
jezoar	1	→
jezrahiah	39	→

Adapted from: [3] (page 157)

Dictionary compression

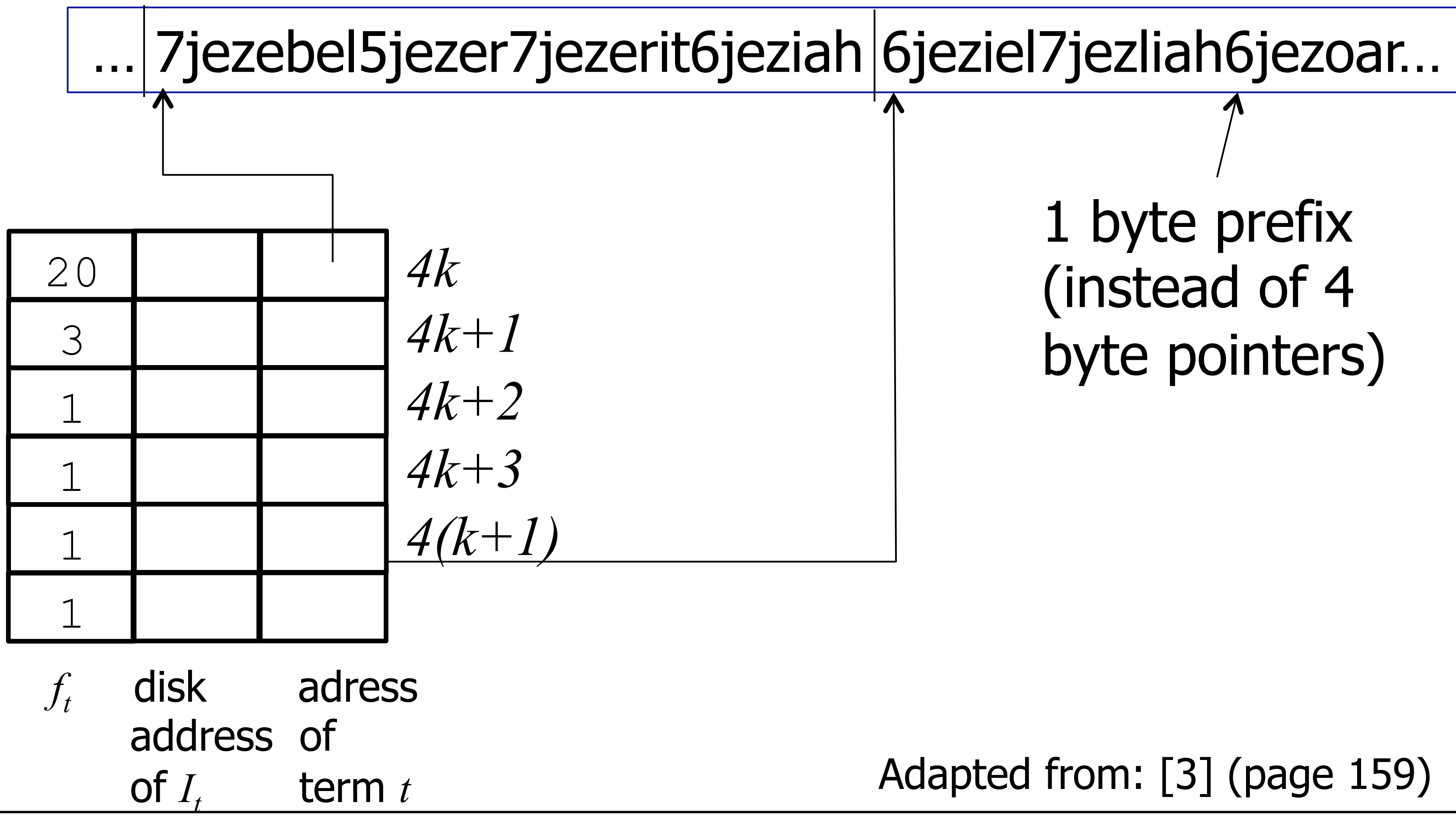
Dictionary-as-a-string



Adapted from: [3] (page 158)

Dictionary compression

Dictionary-as-a-string with reduced term pointers



Adapted from: [3] (page 159)

Dictionary compression

- The efficient single pass indexing approach includes index terms in the runs (not term identifiers)
- Since the terms are processed in lexicographic order, ajdacent terms are likely to have a common prefix
  - Adjacent terms typically share a prefix of 3-5 characters
- Front-coding: instead of storing the term, two integers and a suffix are stored
  1. Number of prefix characters in common with the previous terms
  2. Number of remaining suffix characters when the prefix is removed
  3. Non-matching suffix between consecutive terms

Dictionary compression

Front-coding

- Best explained with an example [3, page 160]:

Term	Complete front coding
jezaniah	
7,**jezebel**	**3,4,ebel**
5,jezer	4,1,r
7,jezerit	5,2,it
6,jeziah	3,3,iah
6,jeziel	4,2,el
7,jezliah	3,4,liah

96 bytes saves 2.5 bytes/word

Dictionary compression

Front-coding

- “Front coding yields a net saving of about 40 percent of the space required for string storage in a typical lexicon of the English language.” [3]

- Problem of complete front-coding: binary search is no longer possible
  - A pointer directly to *4,2,el* will not yield a usable term for binary search

- In practice: every nth term is stored without front coding so that binary search can proceed

Dictionary compression

Front-coding

- “Front coding yields a net saving of about 40 percent of the spa… e Eng…
- Pro… ger pos…
  - … ary
- In … so tha…

Term	Complete front coding	Partial “3-in-4” front coding
jezaniah		
7,**jezebel**	**3,4,ebel**	,7,jezebel
5,jezer	4,1,r	4,1,r
7,jezerit	5,2,it	5,2,it
6,jeziah	3,3,iah	3, ,iah
6,jeziel	4,2,el	,6,jeziel
7,jezliah	3,4,liah	3,4,liah

Distributed indexing

- So far: one machine with limited memory is used to create the index
- Not feasible for very large collections (such as the Web)
  - Index is build by a cluster of machines
  - Several indexers must be coordinated for the final inversion (MapReduce)

- The final index needs to be partitioned, it does not fit into a single machine
  - Splitting the documents across different servers
  - Splitting the index terms across different servers

Distributed indexing

Term-based index partitioning

- Also known as "distributed global indexing"

- Query processing:
  - Queries arrive at the broker which distributes the query and returns the results
  - The broker is in charge of merging the posting lists and producing the final document ranking

- The broker sends requests to the servers containing the query terms; merging occurs in the broker
- Load balancing depends on the distribution of query terms and its co-occurrences
  - Query log analysis is useful, but difficult to get right

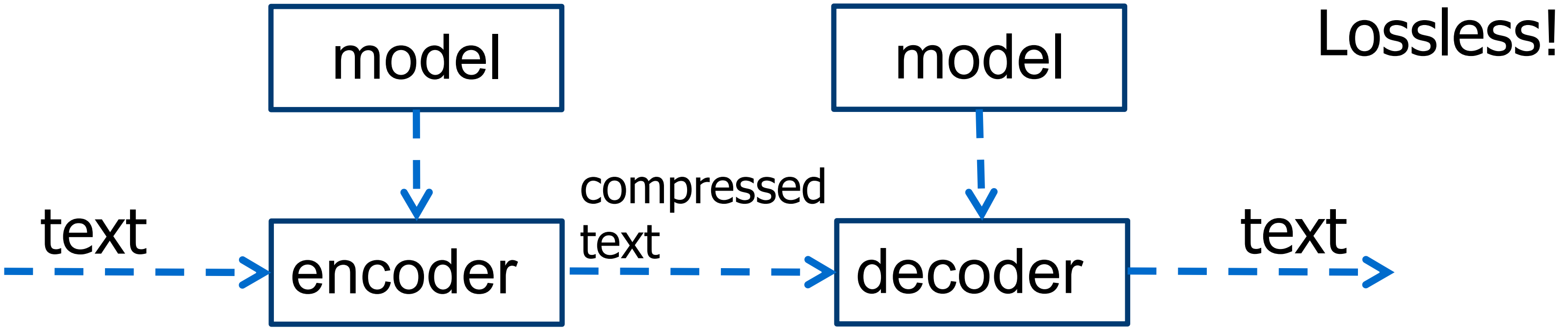
Distributed indexing

Document-based index partitioning

- Also known as “distributed local indexing”
- The common approach for distributed indexing
- Query processing
  - Every server receives all query terms and performs a local search
  - Result documents are sent to the broker, which sorts them
- Issues: maintainance of global collection statistics inside each server (needed for document ranking)

Text compression

- Having looked at inverted file and dictionary compression, lets turn to text compression (document compression)



- 2 classes: **symbolwise** and **dictionary** methods

Symbolwise compression

- **Modeling**: estimation of symbol probabilities (➔statistical methods)
  - Frequently occurring symbols are assigned shorter codewords
  - E.g. in English 'e' is a very common character, 'the' is a common term in most texts, etc.
  - Methods differ in how they estimate the symbol probabilities
    - The more accurate the estimation, the greater the compression
  - Approaches: prediction by partial matching, block sorting, word-based methods, etc.
  - No single best method
- **Coding**: conversion of probabilities into a bitstream
  - Usually based on either Huffman coding or arithmetic coding

Dictionary-based compression

- Achieve compression by replacing words and other fragments of text with an index to an entry in a 'dictionary'
  - Several symbols are represented as one output codeword

- Most significant methods are based on Ziv-Lempel coding
  - Idea: replace strings of characters with a reference to a previous occurrence of the string
  - Effective since most characters can be coded as part of a string that has occurred earlier in the text
    - Compression is achieved if the pointer takes less space than the string it replaces

Models

- *Alphabet*: set of all symbols
- Probability distribution provides an estimated probability for each symbol in the alphabet

- Model provides the probability distribution to the encoder, which uses it to encode the symbol that actually occurs
- The decoder uses an identical model together with the output of the encoder

- Note: encoder cannot boost its probability estimates by looking ahead at the next symbol
  - Decoder and encoder use the same distribution and the decoder cannot look ahead!

Models II

Source coding theorem (Claude Shannon, 1948)

- Information content: numer of bits in which s should be coded (directly related to the predicted probability)

$$I(s) = -\log_2 P(s)$$

- Examples: transmit fair coin toss: $P(head)=0.5$, $-\log_2(0.5)=1$
  transmit u with 2% occurrence: $I(s)=5.6$

- Average amount of information per symbol: entropy H of the probability distribution

$$H = \sum_s P(s) \times I(s) = \sum_s -P(s) \times \log_2 P(s)$$

- H offers a lower bound on compression (source coding theorem)

Models III

- Models can also take preceding symbols into account
  - If ‘q’ was just encountered, the probability of ‘u’ goes up to 95%, based on how often ‘q’ is followed by ‘u’ in a sample text ➔$I(u)=0.074$ bits

- **Finite-context models** of order m take m previous symbols into account

- **Static models**: use the same probability distribution regardless of the text to be compressed
- **Semi-static models**: model generated for each file (requires an initial pass)
  - Model needs to be transmitted to the decoder

Adaptive models

- Adaptive models start with a bland probability distribution and gradually alters it as more symbols are encountered
  - Does not require model transmission to the decoder

- Example: model that uses the previously encoded part of a string as sample to estimate probabilities

- Advantages: robust, reliable and flexible
- Disadvantage: not suitable for random access to files, the decoder needs to process the text from the beginning to build up the correct model

Huffman coding

Huffman 1952

- Coding: determine output representation of a symbol, based on a probability distribution supplied by a model
- Principle: common symbols are coded in few bits, rare symbols are encoded with longer codewords
- Faster than arithmetic coding, achieves less compression

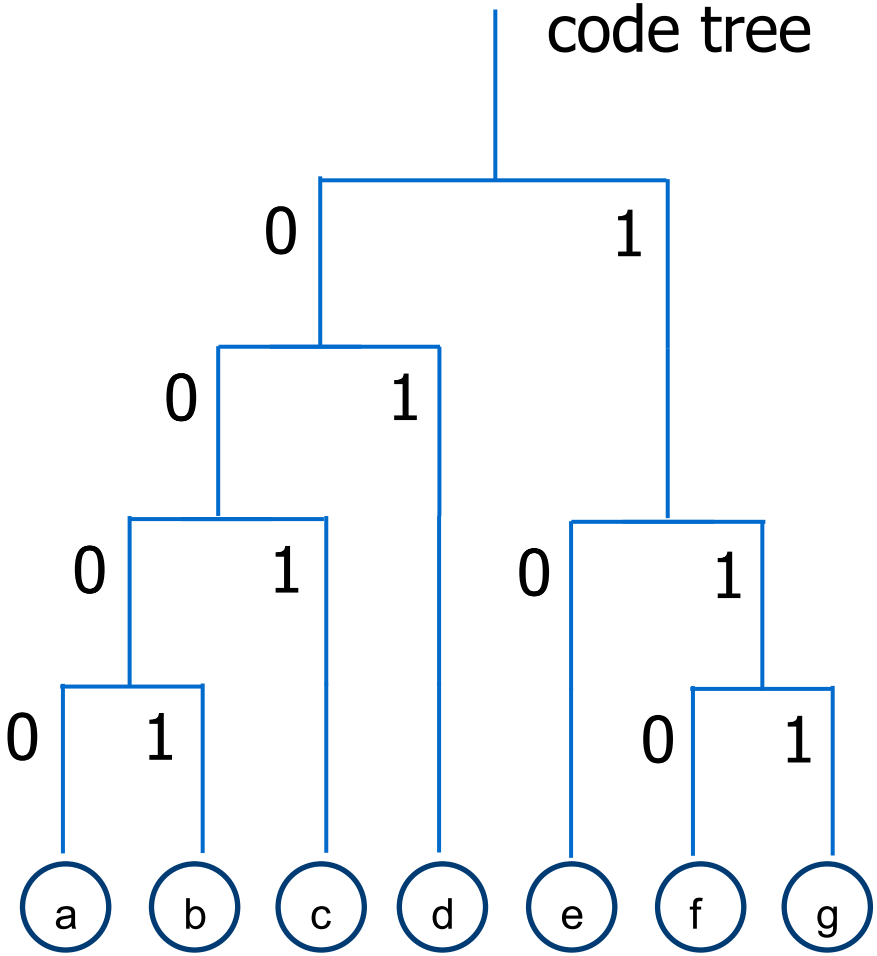
Huffman coding

Principle

Symbol	Codeword	P(s)
a	0000	0.05
b	0001	0.05
c	001	0.1
d	01	0.2
e	10	0.3
f	110	0.2
g	111	0.1

7 symbol alphabet

prefix-free code



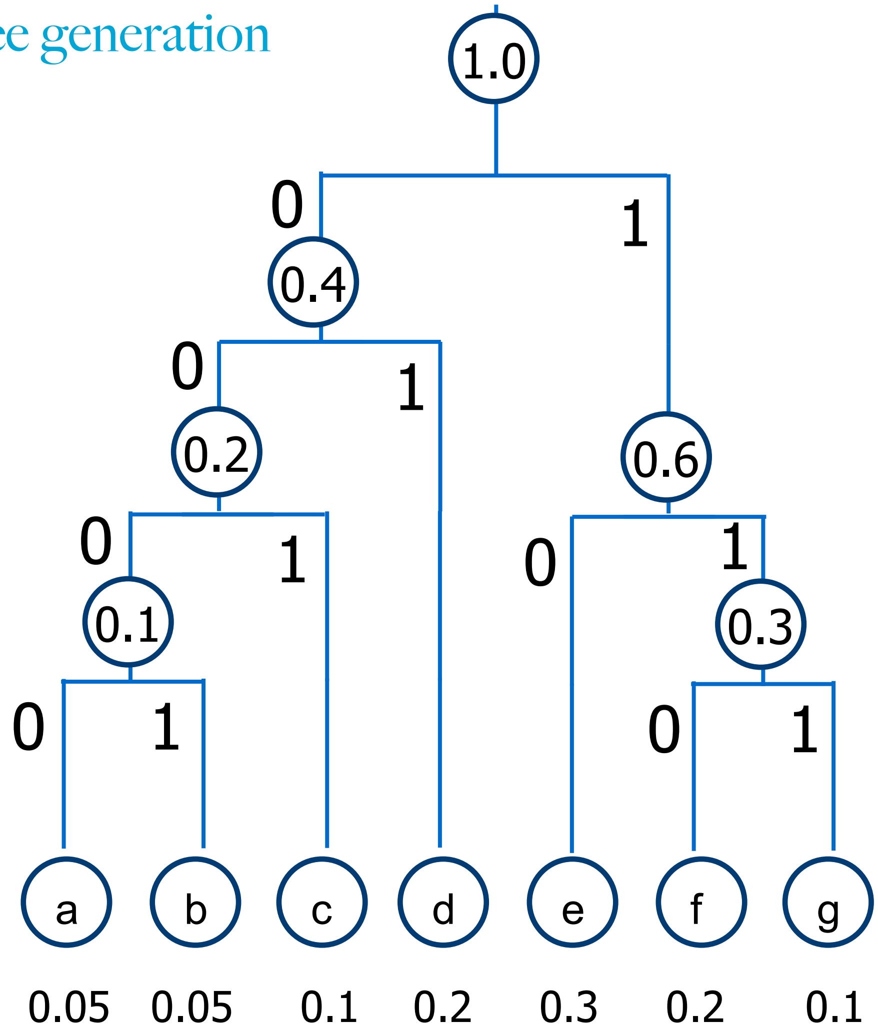
egg → 10111111

deaf → 01100000110

Source: [3]

Huffman coding

Code tree generation



Source: [3]

Huffman coding

Code assignment in pseudocode

① Set T as the set of n singleton sets, each containing one of the n symbols and its probability

② Repeat n-1 times

  ① Set m_1 and m_2: the two subsets of least probability in T

  ② Replace m_1 and m_2 with set $\{m_1, m_2\}$ with $p=P(m_1)+P(m_2)$

③ T now contains only one entry: the root of the Huffman tree

- Considered a good choice for word-based models (rather than character-based)
- Random access is possible (starting points indexed)

Source: [3]

Canonical Huffman code

Code tree not needed for decoding

alphabetically sorted (yopur … zigzag)

Symbol	Length CW	Codeword (CW) bits
yopur	17	00001101010100100
youmg	17	00001101010100101
youthful	17	00001101010100110
zeed	17	00001101010100111
zephyr	17	00001101010101000
zigzag	17	00001101010101001
11th	16	0000110101010101
120	16	0000110101010110
….		
were	8	10100110
which	8	10100111
as	7	1010100
at	7	1010101
For	7	1010110
Had	7	1010111
he	7	1011000
her	7	1011001
His	7	1011010
It	7	1011011
s	7	1011100
…		

- Terms in decreasing order of codeword length
- Within each block of codes of the same length (same freq.), terms are ordered alphabetically

- **Fast encoding**: CW determined from length of CW, how far through the list it is and the CW for the first word of that length
  - 'had' is the 4th seven bit codeword; we know the first seven bit codeword, add 3 (binary) to retrieve 1010111
- **Decoding** without the code tree: list of symbols ordered as described and array storing the first codeword of each distinct length is used instead.

Source: [3]

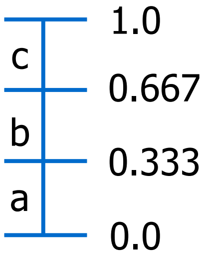
Arithmetic coding

- It can code arbitrarily close to the entropy
  - It is known that it is not possible to code better than the entropy on average
- Huffman coding becomes ineffective when some symbols are highly probable
  - Binary alphabet: $P(s_1)=0.99$ and $P(s_2)=0.01$
  - $I(s_1)=0.015$ bits, though the Huffman coder needs at least one
- Slower than Huffman coding, no easy random access
- **Message is encoded in a real number between 0 and 1**
  - how much data can be encoded in one number depends on the precision of the number

Arithmetic coding

Explained with an example

- Output of an arithmetic coder is a stream of bits
  - Image a "0." in front of the stream and the output becomes a fractional binary between 0 and 1
    - 1010001111 ➔ 0.1010001111 ➔ 0.64 (decimal)



- Compress *bccb* from alphabet *{a,b,c}*
  - Before a part of the message is read: $P(a)=P(b)=P(c)=1/3$ and stored interval boundaries $low=0$ and $high=1$

  ① In each step, narrow the interval to the one corresponding to the character to be encoded: b ➔ $low=0.33$ and $high=0.66$

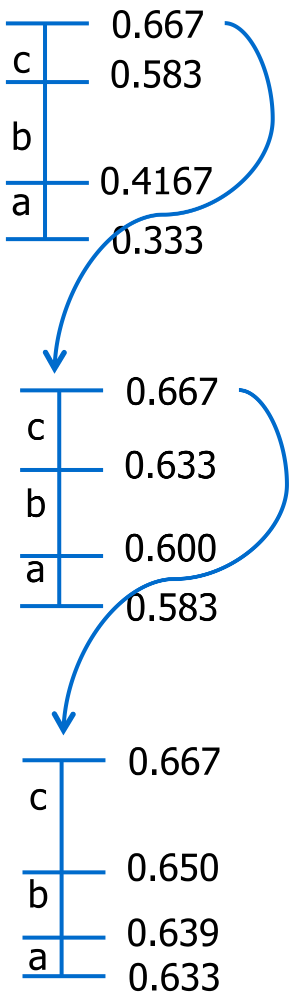
  ② Adapt the probability distribution $P(a)=P(c)=1/4$, $P(b)=2/4$ and redistribute values over reduced interval

Source: [3]

Arithmetic coding

Explained with an example

- Compress *bccb* from alphabet *{a,b,c}*
  - 'b' encoded
    - $P(a)=P(c)=1/4, P(b)=2/4$ and $low=0.33$, $high=0.66$
  - 'c' encoded
    - $P(a)=1/5, P(c)=2/5, P(b)=2/5$ and $low=0.583$ and $high=0.66$
  - 'c' encoded
    - $P(a)=1/6, P(c)=3/6, P(b)=2/6$ and $low=0.633$ and $high=0.667$
  - 'b' encoded

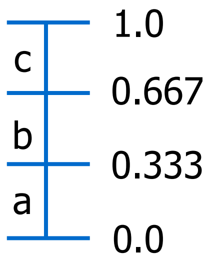


Transmitting **any number** in this interval yields *bccb* (e.g. 0.64*)*

Source: [3]

Arithmetic coding

Explained with an example



- Decompress 0.64 given the alphabet *{a,b,c}*
  - Same uniform probability distribution as in the encoder
  - 0.64 is in the b-interval, thus first codeword is `b'
    - *P(a)=P(c)=1/4, P(b)=2/4* and *low=0.33*, *high=0.66*
  - …

- Compression is achieved because high probability events do not decrease the *low/high* interval a lot, while low probability events result in a much smaller next interval
  - A small final interval requires many digits (bits) to specify a number that is guaranteed to be within the interval
  - A large interval requires few digits

Recommended reading material

- Index compression for information retrieval systems. Roi Blanco Gonzales. PhD thesis. 2008.
  - *http://www.dc.fi.udc.es/~roi/publications/rblanco-phd.pdf*

- Managing Gigabytes: Compressing and Indexing Documents and Images. I.H. Witten, A. Moffat and T.C. Bell. Morgan Kaufmann Publishers. 1999.

- Introduction to Information Retrieval. Manning et al.. Chapters 4&5.

Sources

1. Index compression for information retrieval systems. Roi Blanco Gonzales. PhD thesis. 2008.
2. Efficient document retrieval in main memory. T. Strohman and W.B. Croft. SIGIR 2007.
3. Managing gigabytes. Witten et al., 1999.
4. Efficient single pass indexing. Heinz and Sobel
5. Reviewing records from a gigabyte of text on a mini-computer using statistical ranking.
6. In-situ generation of compressed inverted files. 1995
7. Bell et al. 1993 d-gaps
8. Elias 1975 (gamma/sigma code)
9. Golomb 1966 (golomb code)
10. Introduction to Information Retrieval. Manning et al. 2008