NPFL103: Information Retrieval (3)

Index construction, Distributed and dynamic indexing, Index compression

Pavel Pecina

pecina@ufal.mff.cuni.cz

Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics
Charles University

Original slides are courtesy of Hinrich Schütze, University of Stuttgart.

Contents

Index construction

Hardware basics

- Data access much faster in memory than on HD disk (approx. $10\times$)
- Disk seeks are “idle” time: No data is transferred from disk while the disk head is being positioned.
- To optimize transfer time from disk to memory: one large chunk is faster than many small chunks.
- Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks). Block sizes: 8KB to 256 KB
- Servers used in IR systems typically have tens or hundreds of GBs of RAM, and TBs of disk space.
- Fault tolerance is expensive: It’s cheaper to use many regular machines than one fault tolerant machine.

Some HW statistics

symbol	statistic	value
s	average seek time	5 ms = 5×10^{-3} s
b	transfer time per byte	0.02 μs = 2×10^{-8} s
	processor's clock rate	10^9 s^{-1}
p	lowlevel operation (e.g., compare+swap a word)	0.01 μs = 10^{-8} s
	size of main memory	several GBs
	size of disk space	1 TB or more

- SSD (Solid State Drive) faster but smaller, more expensive, limitted write cycles

RCV1 collection

- Shakespeare's collected works are not large enough for demonstrating many of the points in this course.
- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
- English newswire articles published in 1995–1996 (one year).
- `https://trec.nist.gov/data/reuters/reuters.html`

A Reuters RCV1 document

REUTERS

You are here: Home > News > Science > **Article**

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian

Reuters RCV1 statistics

N	documents	800,000
L	tokens per document	200
M	terms (= word types)	400,000
	bytes per token (incl. spaces/punct.)	6
	bytes per token (without spaces/punct.)	4.5
	bytes per term (= word type)	7.5
T	non-positional postings	100,000,000

Exercise:

1. Average doc. frequency of a term (how many tokens)?
2. 4.5 bytes per token vs. 7.5 bytes per type: why the difference?
3. How many positional postings?

Goal: construct the inverted index

Brutus	⟶	1	2	4	11	31	45	173	174	
Caesar	⟶	1	2	4	5	6	16	57	132	...
Calpurnia	⟶	2	31	54	101					
⋮										

dictionary **postings**

Index construction: Sort postings in memory

term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

$\Longrightarrow$

term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Sort-based index construction

- As we build index, we parse documents one at a time.
- The final postings for any term are incomplete until the end.
- Can we keep all postings in memory and then do the sort in-memory at the end?
- No, not for large collections
- At 10–12 bytes per postings entry, we need a lot of space for large collections.
- $T = 100{,}000{,}000$ in the case of RCV1: we can do this in memory on a typical current machine.
- In-memory index construction does not scale for large collections.
- Thus: We need to store intermediate results on disk.

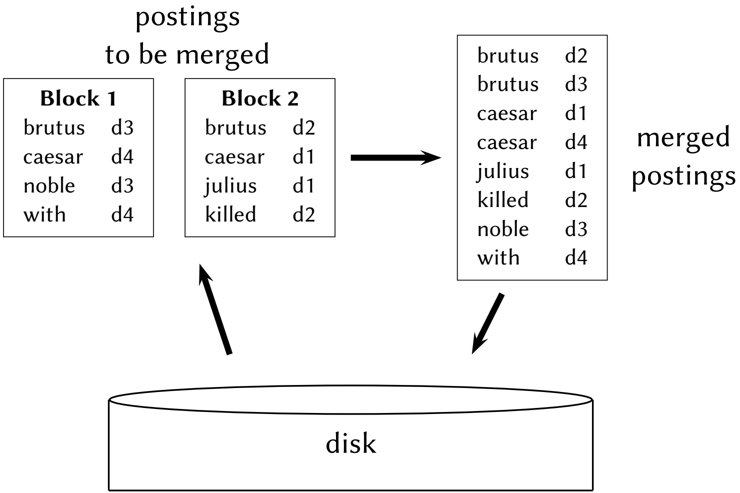
Same algorithm for disk?

- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?
- No: Sorting $T = 100{,}000{,}000$ records on disk is too slow – too many disk seeks.
- We need an external sorting algorithm.

"External" sorting algorithm (using few disk seeks)

- We must sort $T = 100{,}000{,}000$ non-positional postings.
  - Each posting has size 12 bytes (4+4+4: termID, docID, doc. freq).
- Define a block to consist of $10{,}000{,}000$ such postings
  - We can easily fit that many postings into memory.
  - We will have 10 such blocks for RCV1.
- Basic idea of algorithm:
  - For each block:
    (i) accumulate postings, (ii) sort in memory, (iii) write to disk
  - Then merge the blocks into one long sorted order.

Merging two blocks



Blocked Sort-Based Indexing (BSBI)

BSBIndexConstruction()

$n \leftarrow 0$

while (all documents have not been processed)

do $n \leftarrow n + 1$

    $block \leftarrow$ ParseNextBlock()

    BSBI-Invert($block$)

    WriteBlockToDisk($block, f_n$)

MergeBlocks($f_1, \ldots, f_n; f_{\text{merged}}$)

- BSBI-Invert:
  1. sort [termID, docID] pairs
  2. collect [termID, docID] pairs with the same docID
- Key decision: What is the size of one block?

Problem with sort-based algorithm

- Our assumption was: we can keep the dictionary in memory.
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.
- Actually, we could work with [term, docID] postings instead of [termID, docID] postings ...
- ...but then intermediate files become very large. (We would end up with a scalable, but very slow index construction method.)

Single-pass in-memory indexing (SPIMI)

- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.
- Key idea 2: Don't sort. Accumulate postings in postings lists as they occur.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.

SPIMI-Invert

```
SPIMI-Invert(token_stream)
  output_file ← NewFile()
  dictionary ← NewHash()
  while (free memory available)
  do token ← next(token_stream)
     if term(token) ∉ dictionary
        then postings_list ← AddToDictionary(dictionary,term(token))
        else postings_list ← GetPostingsList(dictionary,term(token))
     if full(postings_list)
        then postings_list ← DoublePostingsList(dictionary,term(token))
     AddToPostingsList(postings_list,docID(token))
  sorted_terms ← SortTerms(dictionary)
  WriteBlockToDisk(sorted_terms,dictionary,output_file)
  return output_file
```

- Merging of blocks is analogous to BSBI.
- Compression of terms/postings makes SPIMI even more efficient

Distributed indexing

Distributed indexing

- For web-scale indexing: must use a distributed computer cluster
- Individual machines are fault-prone: can unpredictably slow down or fail.
- How do we exploit such a pool of machines?

Google data centers (estimates from 2016, Gartner)

- Google data centers mainly contain commodity machines.
- 2.5 million servers in 15 data centers are distributed all over the world.
- This is about 10% of the computing capacity of the world!
- If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the system (assuming it does not tolerate failures)?

  Answer: 37% ($0.999^{1000} = 0.3677$)
- Suppose a server will fail after 3 years. For an installation of 1 million servers, what is the interval between machine failures?

  Answer: <2 minutes ($(3 * 365 * 24 * 60)/1000000 = 1.5768$)

Distributed indexing

- Maintain a master machine directing the job – considered “safe”
- Break up indexing into sets of parallel tasks
- Master machine assigns each task to an idle machine from a pool.

Parallel tasks

- We will define two sets of parallel tasks and deploy two types of machines to solve them: parsers, inverters
- Break the input document collection into splits (corresponding to blocks in BSBI/SPIMI)
- Each split is a subset of documents.

Process

Master:

1. Assigns a split to an idle parser machine.

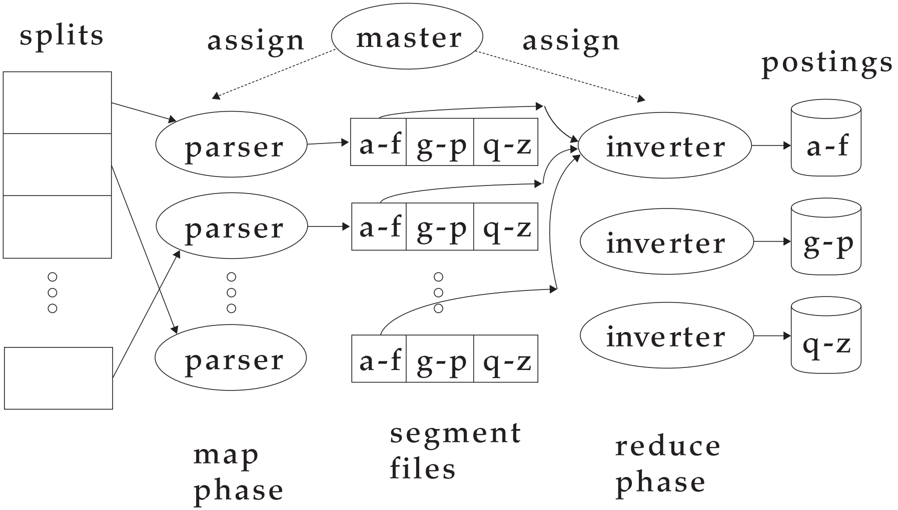
Parser:

1. Reads a document at a time and emits [term,docID]-pairs.
2. Writes pairs into j partitions each for a range of terms' first letters (e.g., a–f, g–p, q–z; here: $j = 3$).

Inverter:

1. Collects all [term,docID] pairs (= postings) for one term-partition (e.g., for a–f).
2. Sorts and writes to postings lists

Data flow


splits
assign
master
assign
postings
parser
a-f g-p q-z
inverter
a-f
parser
a-f g-p q-z
inverter
g-p
parser
a-f g-p q-z
inverter
q-z
map
phase
segment
files
reduce
phase

MapReduce

- The index construction algorithm we just described is an instance of MapReduce.
- MapReduce is a robust and conceptually simple framework for distributed computing ...
- ...without having to write code for the distribution part.
- The original Google indexing system consisted of a number of phases, each implemented in MapReduce.

Dynamic indexing

Dynamic indexing

- Up to now, we have assumed that collections are static.
- They rarely are: Documents are inserted, deleted and modified.
- Dictionary and postings lists have to be dynamically modified.

Dynamic indexing: Simplest approach

- Maintain big main index on disk
- New docs go into small auxiliary index in memory.
- Search across both, merge results
- Periodically, merge auxiliary index into big index
- Deletions:
  - Invalidation bit-vector for deleted docs
  - Filter docs returned by index using this bit-vector

Issue with multiple indexes

- Corpus-wide statistics are hard to maintain.
- E.g., for hit-based spelling correction: how do we determine which correction has the most hits in the collection?
- We will see that other such statistics are important in ranking.
- There is no easy way around this if we want to do dynamic indexing efficiently.

Issue with auxiliary and main index

- Frequent merges
- Poor search performance during index merge
- Actually:
  - Merging of the auxiliary index into the main index is not that costly if we keep a separate file for each postings list.
  - But then we would need a lot of files – inefficient.
- Assumption for the rest of the lecture: The index is one big file.
- In reality: Use a scheme somewhere in between (e.g., split very large postings lists into several files, collect small postings lists in one file)

Logarithmic merge

- Logarithmic merging amortizes cost of merging indexes over time.

  $\rightarrow$ Users see smaller effect on response times.
- Maintain a series of indexes, each twice as large as the previous one.
- Keep smallest (Z_0) in memory
- Larger ones (I_0, I_1, ...) on disk
- If Z_0 gets too big ($> n$), write to disk as I_0

  ... or merge with I_0 (if I_0 already exists) and write merger to I_1 etc.

LMergeAddToken(*indexes*, Z_0, *token*)

$Z_0 \leftarrow$ Merge(Z_0, {*token*})
if $|Z_0| = n$
  then for $i \leftarrow 0$ **to** ∞
    do if $I_i \in$ *indexes*
      then $Z_{i+1} \leftarrow$ Merge(I_i, Z_i)
        (Z_{i+1} is a temporary index on disk.)
        indexes $\leftarrow$ *indexes* $-$ {I_i}
      else $I_i \leftarrow Z_i$ *(Z_i becomes the permanent index I_i.)*
        indexes $\leftarrow$ *indexes* $\cup$ {I_i}
        Break
    $Z_0 \leftarrow \emptyset$

LogarithmicMerge()

$Z_0 \leftarrow \emptyset$ *(Z_0 is the in-memory index.)*
indexes $\leftarrow \emptyset$
while true
do LMergeAddToken(*indexes*, Z_0, getNextToken())

Logarithmic merge

- Number of indexes bounded by $O(\log T)$ (T is total number of postings read so far)
- So query processing requires the merging of $O(\log T)$ indexes
- Time complexity of index construction is $O(T \log T)$.

  ...because each of T postings is merged $O(\log T)$ times.

- Auxiliary index: index construction time is $O(T^2)$ as each posting is touched in each merge.
  - Suppose auxiliary index has size a
  - $a + 2a + 3a + 4a + \ldots + na = a\frac{n(n+1)}{2} = O(n^2)$
- So logarithming merging is an order of magnitude more efficient.

Dynamic indexing at large search engines

Often a combination of:

1. Frequent incremental changes
2. Rotation of large parts of the index that can then be swapped in
3. Occasional complete rebuild (becomes harder with increasing size)

Building positional indexes

- Basically the same problem except that the intermediate data structures are large.

Index compression

Inverted index

For each term *t*, we store a list of all documents that contain *t*.

BRUTUS	$\longrightarrow$	1	2	4	11	31	45	173	174	
CAESAR	$\longrightarrow$	1	2	4	5	6	16	57	132	...
CALPURNIA	$\longrightarrow$	2	31	54	101					
$\vdots$										

dictionary **postings file**

- How much space do we need for the dictionary?
- How much space do we need for the postings file?
- How can we compress them?

Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Speed up transferring data from disk to memory (increases speed)

[read compressed data and decompress in memory]
is faster than
[read uncompressed data]

- Premise: Decompression algorithms are fast.
  … this is true of the decompression algorithms we will use.

Why compression in information retrieval?

- First, we will consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory
- Then for the postings file
  - Motivation: reduce disk space needed, decrease time needed to read from disk
  - Note: Large search engines keep significant part of postings in memory
- We will use various compression schemes for dictionary and postings.

Lossy vs. lossless compression

- Lossy compression: Discard some information
- Several of the preprocessing steps we frequently use can be viewed as lossy compression:
  - lowercasing, stop words removal, stemming, number elimination
- Lossless compression: All information is preserved.
  - What we mostly do in index compression

Model collection: The Reuters collection

symbol	statistic	value
N	documents	800,000
L	avg. # word tokens per document	200
M	word types	400,000
	avg. # bytes per word token (incl. spaces/punct.)	6
	avg. # bytes per word token (without spaces/punct.)	4.5
	avg. # bytes per word type	7.5
T	non-positional postings	100,000,000

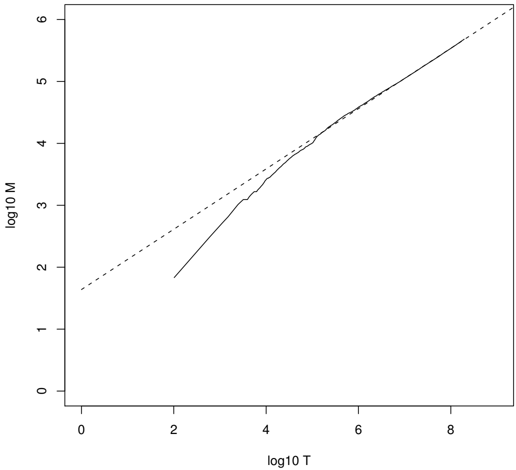
Effect of preprocessing for Reuters

size of	word types dictionary			non-positional postings non-positional index			positional postings positional index		
	size	$\Delta\%$	$\sum\Delta\%$	size	Δ	$\sum\Delta\%$	size	Δ	$\sum\Delta\%$
unfiltered	484,494			109,971,179			197,879,290		
no numbers	473,723	-2%	-2%	100,680,242	-8%	-8%	179,158,204	-9%	-9%
case folding	391,523	-17%	-19%	96,969,056	-3%	-12%	179,158,204	-0%	-9%
30 stop w's	391,493	-0%	-19%	83,390,443	-14%	-24%	121,857,825	-31%	-38%
150 stop w's	391,373	-0%	-19%	67,001,847	-30%	-39%	94,516,599	-47%	-52%
stemming	322,383	-17%	-33%	63,812,300	-4%	-42%	94,516,599	-0%	-52%

How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- Not really: At least $70^{20} \approx 10^{37}$ different words of length 20.
- The vocabulary will keep growing with collection size.
- Heaps' law: $M = kT^b$
  - Empirical law.
  - M – size of the vocabulary, T – number of tokens in the collection.
  - Linear in log-log space.
  - Typical values for the parameters: $30 \leq k \leq 100$ and $b \approx 0.5$.

Heaps' law for Reuters



Vocabulary size M is a function of collection size T:

$$M = kT^b$$

The best least squares fit for Reuters RCV1:

$$\log_{10} M = 0.49 * \log_{10} T + 1.64$$

$M = 10^{1.64} T^{0.49}$
$k = 10^{1.64} \approx 44$
$b = 0.49$.

Empirical fit for Reuters

- Good, as we just saw in the graph.
- For the first 1,000,020 tokens Heaps'law predicts 38,323 terms:

$$44 \times 1{,}000{,}020^{0.49} \approx 38{,}323$$

- The actual number is 38,365 terms, very close to the prediction.
- Empirical observation: fit is good in general.

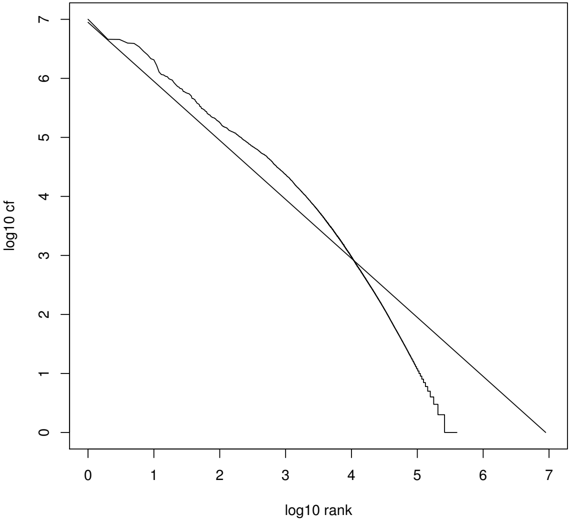
Zipf's law

- We have characterized the growth of the vocabulary in collections.
- We also want to know how many frequent vs. infrequent terms we should expect in a collection.
- In natural language, there are a few very frequent terms and very many very rare terms.
- Zipf's law: $\mathrm{cf}_i \propto \frac{1}{i}$
- The i^{th} most frequent term has frequency cf_i proportional to $1/i$.
- Collection frequency cf_i: number of occurrences of term t_i in the collection.

Zipf's law: example

- Zipf's law: $\mathrm{cf}_i \propto \frac{1}{i}$
- The i^{th} most frequent term has frequency cf_i proportional to $1/i$.
- So if the most frequent term (*the*) occurs cf_1 times, then the second most frequent term (*of*) has half as many occurrences $\mathrm{cf}_2 = \frac{1}{2}\mathrm{cf}_1$ ...
- ...and the third most frequent term (*and*) has a third as many occurrences $\mathrm{cf}_3 = \frac{1}{3}\mathrm{cf}_1$ etc.
- Equivalent: $\mathrm{cf}_i = ci^k$ and $\log \mathrm{cf}_i = \log c + k \log i$ (for $k = -1$)
- Example of a power law.

Zipf's law for Reuters



Fit is not great. What is important is the key insight:

Few frequent terms, many rare terms.

Dictionary compression

- The dictionary is small compared to the postings file.
- But we want to keep it in memory.
- Also: competition with other applications, cell phones, onboard computers, fast startup time
- So compressing the dictionary is important.

Recall: Dictionary as array of fixed-width entries

Dictionary:

term	document frequency	pointer to postings list
a	656,265	$\longrightarrow$
aachen	65	$\longrightarrow$
...	...	...
zulu	221	$\longrightarrow$

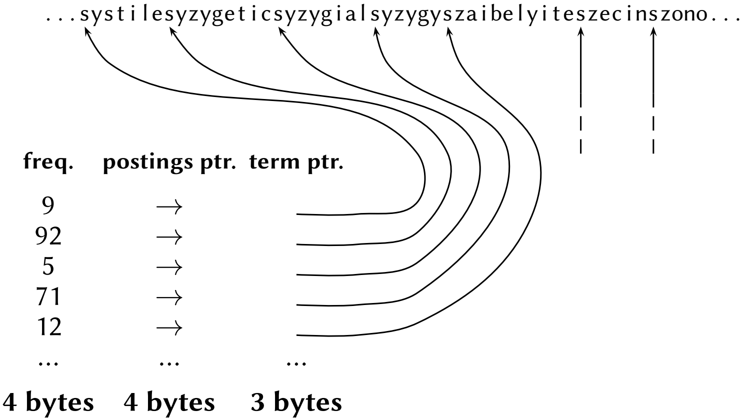
space needed: 20 bytes 4 bytes 4 bytes

Space for Reuters: (20+4+4)*400,000 = 11.2 MB

Fixed-width entries are bad.

- Most of the bytes in the term column are wasted.
  - We allot 20 bytes for terms of length 1.
- We can't handle HYDROCHLOROFLUOROCARBONS and SUPERCALIFRAGILISTICEXPIALIDOCIOUS
- Average length of a term in English: 8 characters
- How can we use on average 8 characters per term?

Dictionary as a string



Space for dictionary as a string

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string
- 3 bytes per pointer into string
  (need $\log_2 8 \cdot 400000 < 24$ bits to resolve $8 \cdot 400{,}000$ positions)
- Space: $400{,}000 \times (4 + 4 + 3 + 8) = 7.6\text{MB}$ (compared to 11.2 MB for fixed-width array)

Dictionary as a string with blocking

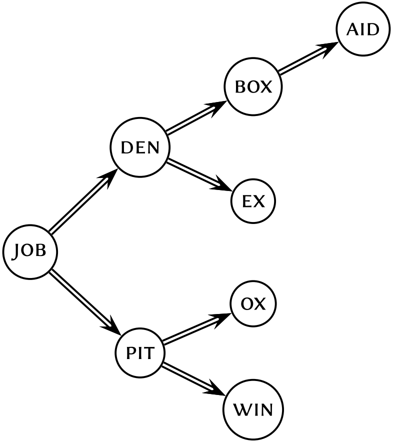
. . . 7systile9syzygetic8syzygial6syzygy11szaibelyite6szecin. . .

freq.	**postings ptr.**	**term ptr.**
9	$\rightarrow$	
92	$\rightarrow$	
5	$\rightarrow$	
71	$\rightarrow$	
12	$\rightarrow$	
...	...	...

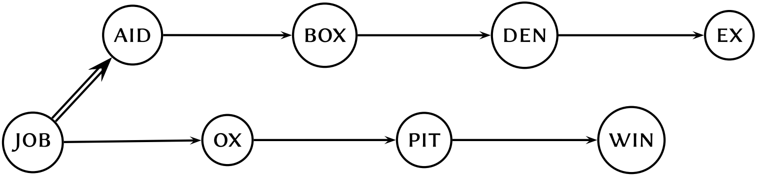
Space for dictionary as a string with blocking

- Example block size $k = 4$
- Where we used 4×3 bytes for term pointers without blocking ...
- ...we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.
- We save $12 - (3 + 4) = 5$ bytes per block.
- Total savings: $400{,}000/4 * 5$ = 0.5 MB
- This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

Lookup of a term without blocking



Lookup of a term with blocking: (slightly) slower



Front coding

One block in blocked compression ($k = 4$) ...

8 a u t o m a t a **8** a u t o m a t e **9** a u t o m a t i c **10** a u t o m a t i o n

$\Downarrow$

...further compressed with front coding.

8 a u t o m a t $*$ a **1** $\diamond$ e **2** $\diamond$ i c **3** $\diamond$ i o n

Dictionary compression for Reuters: Summary

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, with blocking & front coding	5.9

Postings compression

- The postings file is much larger than the dictionary (factor >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 $\log_2 800{,}000 \approx 19.6 < 20$ bits per docID.
- Our goal: use a lot less than 20 bits per docID.

Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: COMPUTER: 283154, 283159, 283202, ...
- It suffices to store gaps: 283159-283154=5, 283202-283154=43
- Example postings list using gaps : COMPUTER: 283154, 5, 43, ...
- Gaps for frequent terms are small.
- Thus: We can encode small gaps with fewer than 20 bits.

Gap encoding

	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		248100								

Compression of Reuters

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, with blocking & front coding	5.9
collection (text, xml markup etc)	3600.0
collection (text)	960.0
T/D incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0

Term-document incidence matrix

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
ANTHONY	1	1	0	0	0	1	
BRUTUS	1	1	0	1	0	0	
CAESAR	1	1	0	1	1	1	
CALPURNIA	0	1	0	0	0	0	
CLEOPATRA	1	0	0	0	0	0	
MERCY	1	0	1	1	1	1	
WORSER	1	0	1	1	1	0	
...							

Entry 1 if term occurs. e.g. CALPURNIA occurs in *Julius Caesar*.
Entry 0 if term doesn't occur. e.g. CALPURNIA doesn't occur in *The tempest*.

Summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient.
- Only 10-15% of the total size of the text in the collection.
- However, we've ignored positional and frequency information.
- For this reason, space savings are less in reality.