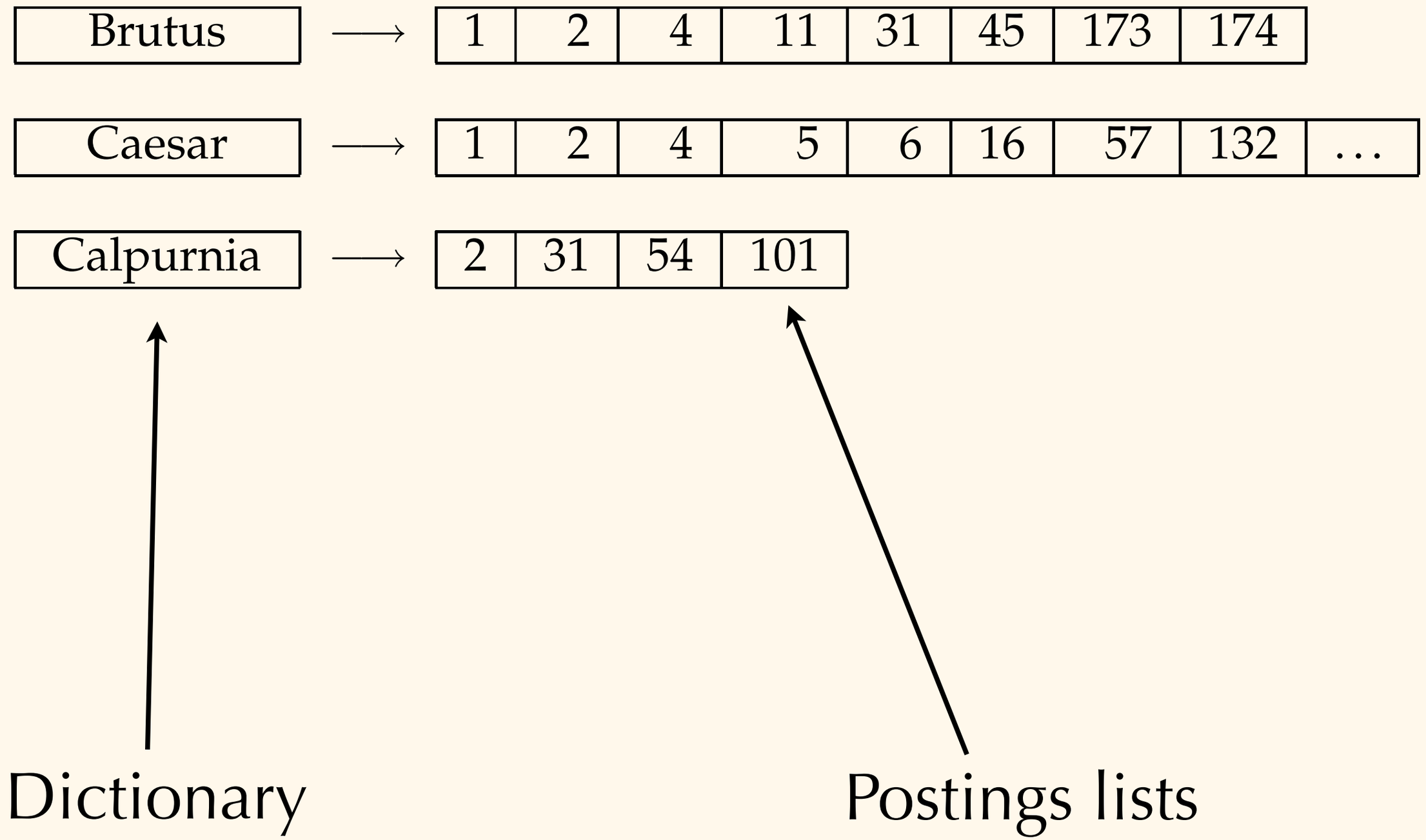
Index Construction & Compression: Agenda

- Practical considerations
- Building indices
  - Static indexing approaches
  - Dynamic indexing
- Storing indices
  - Dictionary compression
  - Posting list compression

Our friend, the inverted index:



Basic steps to for building an index:

1. Pass through collection, pair terms and docIDs
2. Group docIDs by term
3. Convert <term, docID> tuples to <term, [docID...]> tuples; calculate other misc. statistics

Translation: Lots of sorting! By term, docID, etc.

When the collection can fit in memory, this is very simple...

One measurement motivates most index construction & compression techniques:

Statistic	Value
average seek time	$5\ \text{ms} = 5 \times 10^{-3}\ \text{s}$
transfer time per byte	$0.02\ \mu\text{s} = 2 \times 10^{-8}\ \text{s}$
processor's clock rate	$10^9\ \text{s}^{-1}$
lowlevel operation (e.g., compare & swap a word)	$0.01\ \mu\text{s} = 10^{-8}\ \text{s}$

$$10^{-3} \ggg 10^{-8}$$

The central idea:

If we can't fit everything in memory...

... we'll need to use a disk-based *external sorting algorithm*...

... and do it in such a way as to minimize disk seeks.

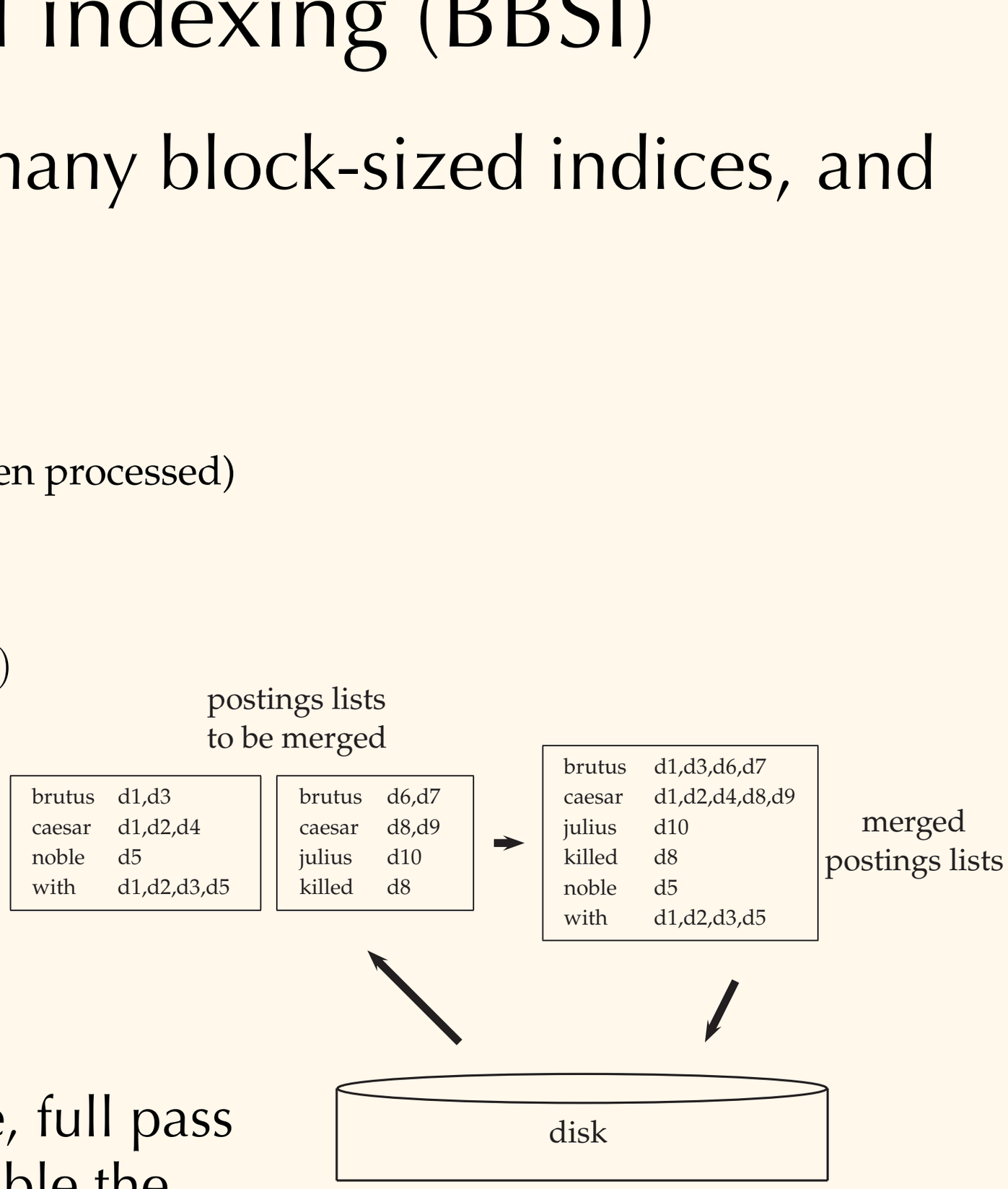
Disks store data in contiguous chunks, or "blocks"...

... and that's how operating systems get data from disks.

Blocked sort-based indexing (BBSI)

The basic idea: make many block-sized indices, and then merge them.

BSBIndexConstruction() *
1 $n \leftarrow 0$
2 **while** (all documents have not been processed)
3 **do** $n \leftarrow n + 1$
4 $block \leftarrow$ ParseNextBlock()
5 BSBI-Invert($block$)
6 WriteBlockToDisk($block, f_n$)
7 MergeBlocks($f_1, \ldots, f_n; f_{\text{merged}}$)



* We also have to do a separate, full pass through the collection to assemble the dictionary and compute termIDs.

Blocked sort-based indexing (BBSI)

BBSI has an important limitation:

Even though the postings are split up by block size...
... the dictionary is not.

We still must maintain a term->termID data structure that is shared by all blocks, and this might not fit in memory.

Single-pass in-memory indexing (SPIMI)

The basic idea: make many *independent* block-sized indices, and then merge them.

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6        then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

Key difference: uses raw terms instead of shared termIDs, so each block has its own dictionary.

Also: lower overhead, so larger blocks can be processed.

Distributed Indexing:

For very large collections, it may make sense to distribute indexing across multiple computers.

Map-Reduce is a common distributed-computing paradigm.

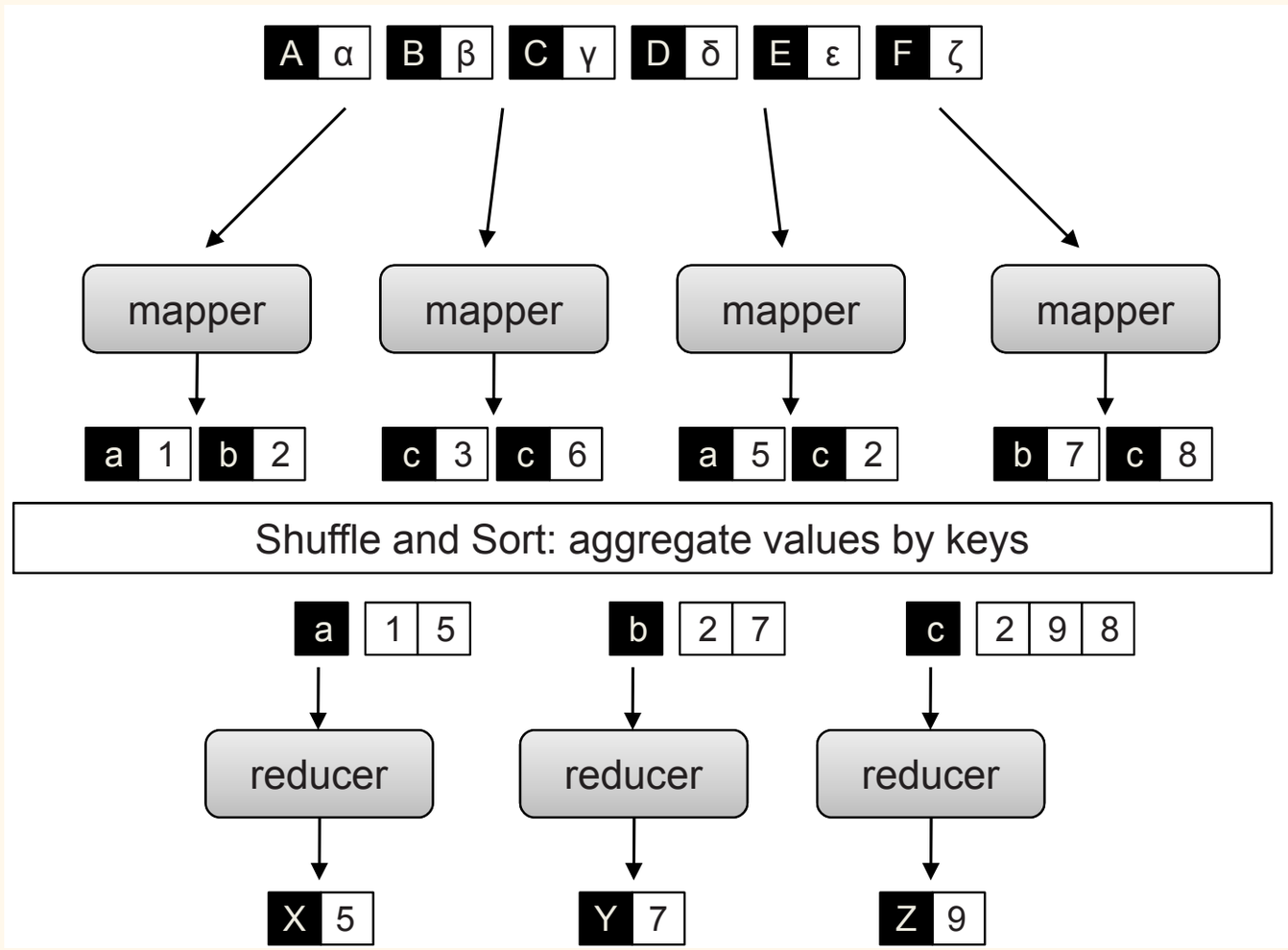


Figure from Lin & Dyer 2010.

Distributed Indexing:

```
1: class Mapper
2:     procedure Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)
```

```
1: class Reducer
2:     procedure Reduce(term t, postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...])
3:         P ← new List
4:         for all posting ⟨a, f⟩ ∈ postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...] do
5:             P.Add(⟨a, f⟩)
6:         P.Sort()
7:         Emit(term t, postings P)
```

Figure from Lin & Dyer 2010.

Distributed Indexing:

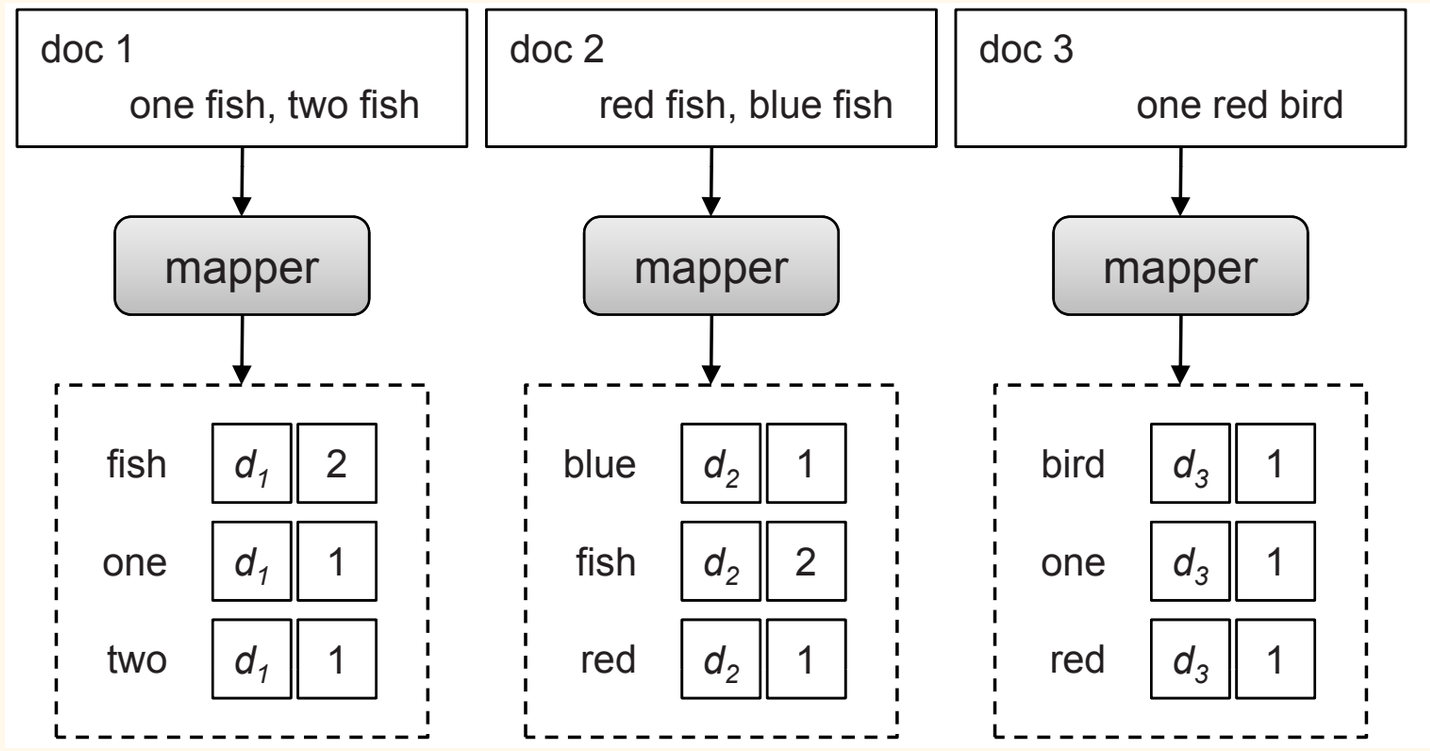


Figure from Lin & Dyer 2010.

Distributed Indexing:

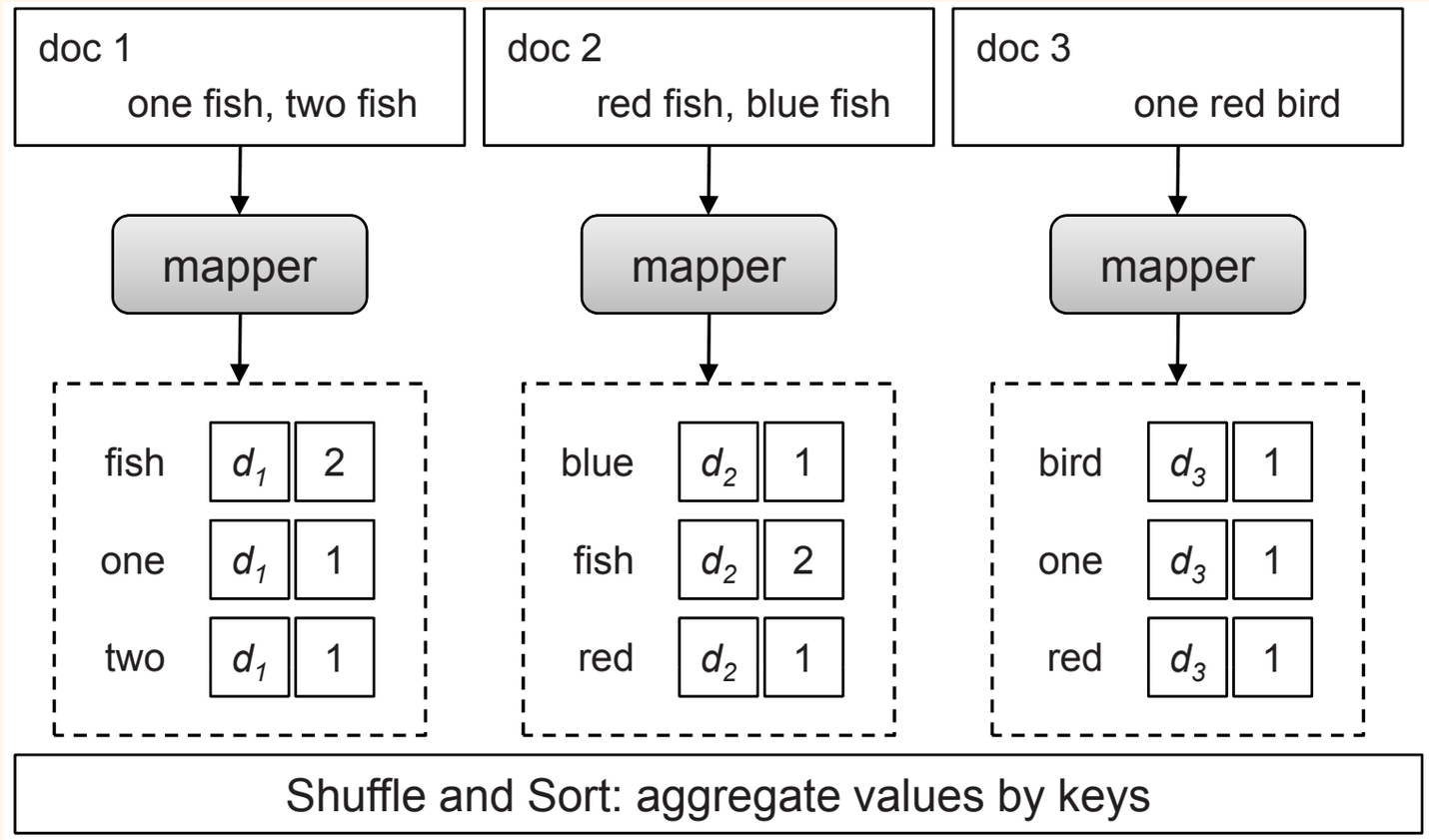


Figure from Lin & Dyer 2010.

Distributed Indexing:

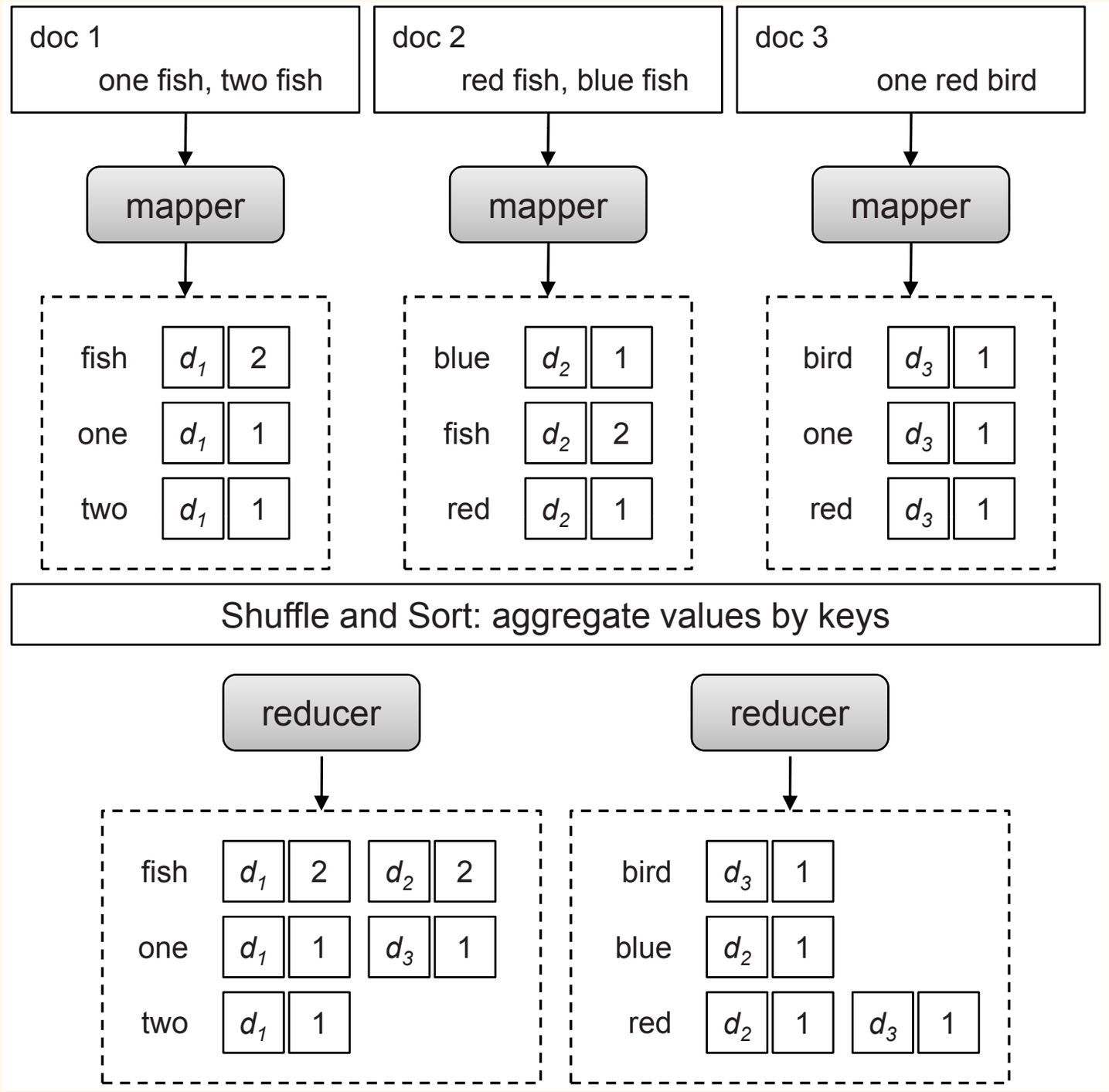


Figure from Lin & Dyer 2010.

Distributed Indexing:

```
1: class Mapper
2:     method Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(tuple ⟨t, n⟩, tf H{t})

1: class Reducer
2:     method Initialize
3:         t_prev ← ∅
4:         P ← new PostingsList
5:     method Reduce(tuple ⟨t, n⟩, tf [f])
6:         if t ≠ t_prev ∧ t_prev ≠ ∅ then
7:             Emit(term t, postings P)
8:             P.Reset()
9:         P.Add(⟨n, f⟩)
10:        t_prev ← t
11:    method Close
12:        Emit(term t, postings P)
```

Figure from Lin & Dyer 2010.

What happens when new data needs to be added to an index?

1. Maintain an “auxiliary index” containing the new data, query both, and merge periodically;
2. Build a second full index periodically and “switch over” when it’s done.

Option 1 is attractive but complex; option 2 is less flexible and expensive but is simpler.

How to represent auxiliary index?

The easiest way is as a large collection of posting files- then, merging is just a simple append operation.

However, most file systems don't appreciate having millions of files (also disk seek time, etc.).

So, the tradeoff is: for merge speed, we want as small an auxiliary index as possible...

... but large enough to not run into storage-related complications; also, we want to minimize merges.

Also, the naïve approach results in overall $O(T^2)$ index construction time (because each posting list has to be merged in each merge).

Can we do better?

Solution: Logarithmic merging.

- Maintain a series of indexes, each twice as large as the previous one
  - At any time, some of these powers of 2 are instantiated
- 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) as Z_1
- Either write merge Z_1 to disk as I_1 (if no I_1)
- Or merge with I_1 to form Z_2

Taken from Manning, et al.'s slides on the subject.

Solution: Logarithmic merging.

Index construction is now O(TlogT) on average, since each posting is only merged logT times...

But query performance just went down: we have to merge log T indices to deliver results.

Also, it is now much harder to maintain collection-wide statistics (needed for spelling suggestion, result ranking, etc.).

What about positional indexes?

General process is similar...

... But storage needs are much greater (each posting contains add'l metadata, etc.)...

Other tricks:

Ordering of postings: Newest first? Oldest first? "Impact-ranked"?

Security: Including ACL information in index?

Index Construction & Compression: Agenda

- Practical considerations
- Building indices
  - Static indexing approaches
    - Dynamic indexing
- Storing indices
  - Dictionary compression
  - Posting list compression

Why compress?

The obvious answer: to save disk space.

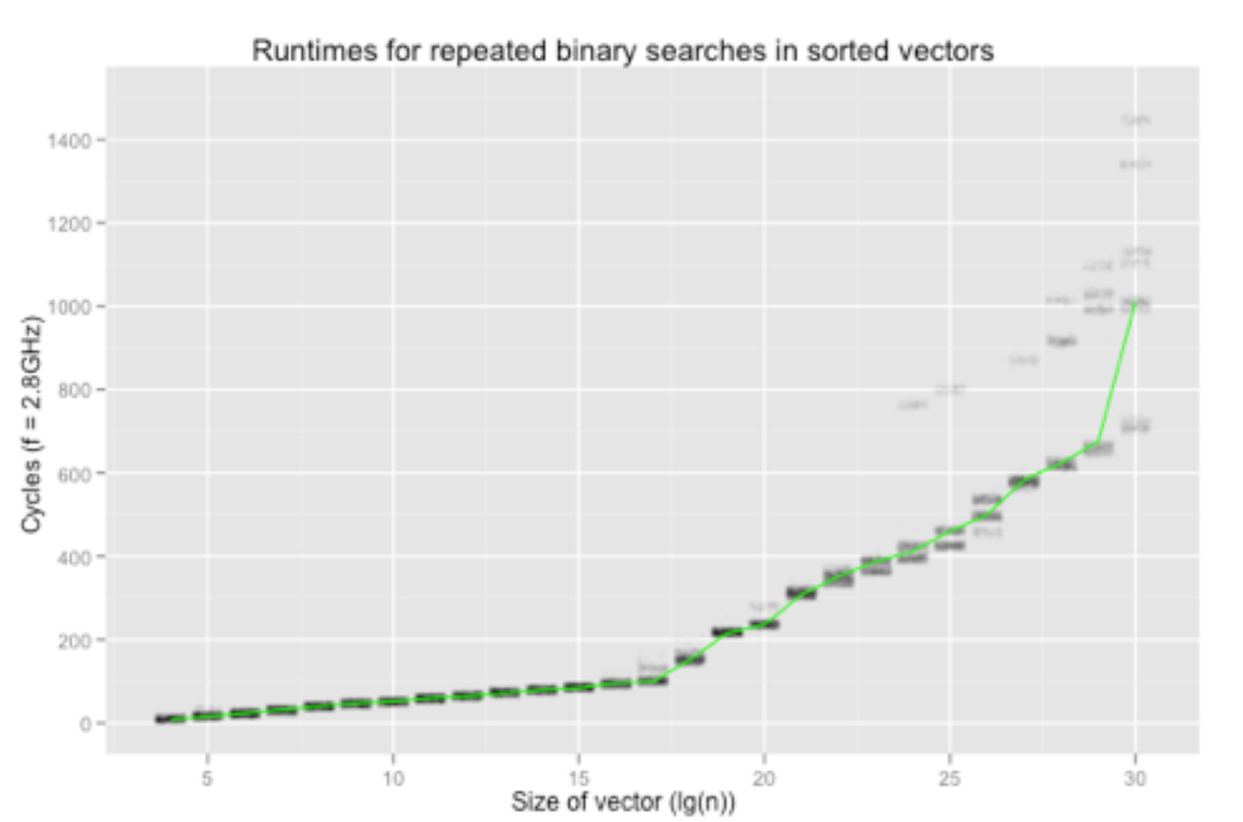
A less obvious answer: to keep more data in the computer's cache.

Statistic	Value
average seek time	$5\ \text{ms} = 5 \times 10^{-3}\ \text{s}$
transfer time per byte	$0.02\ \mu\text{s} = 2 \times 10^{-8}\ \text{s}$
processor's clock rate	$10^9\ \text{s}^{-1}$
lowlevel operation (e.g., compare & swap a word)	$0.01\ \mu\text{s} = 10^{-8}\ \text{s}$

We can decompress data much faster than the disk can get it to us!

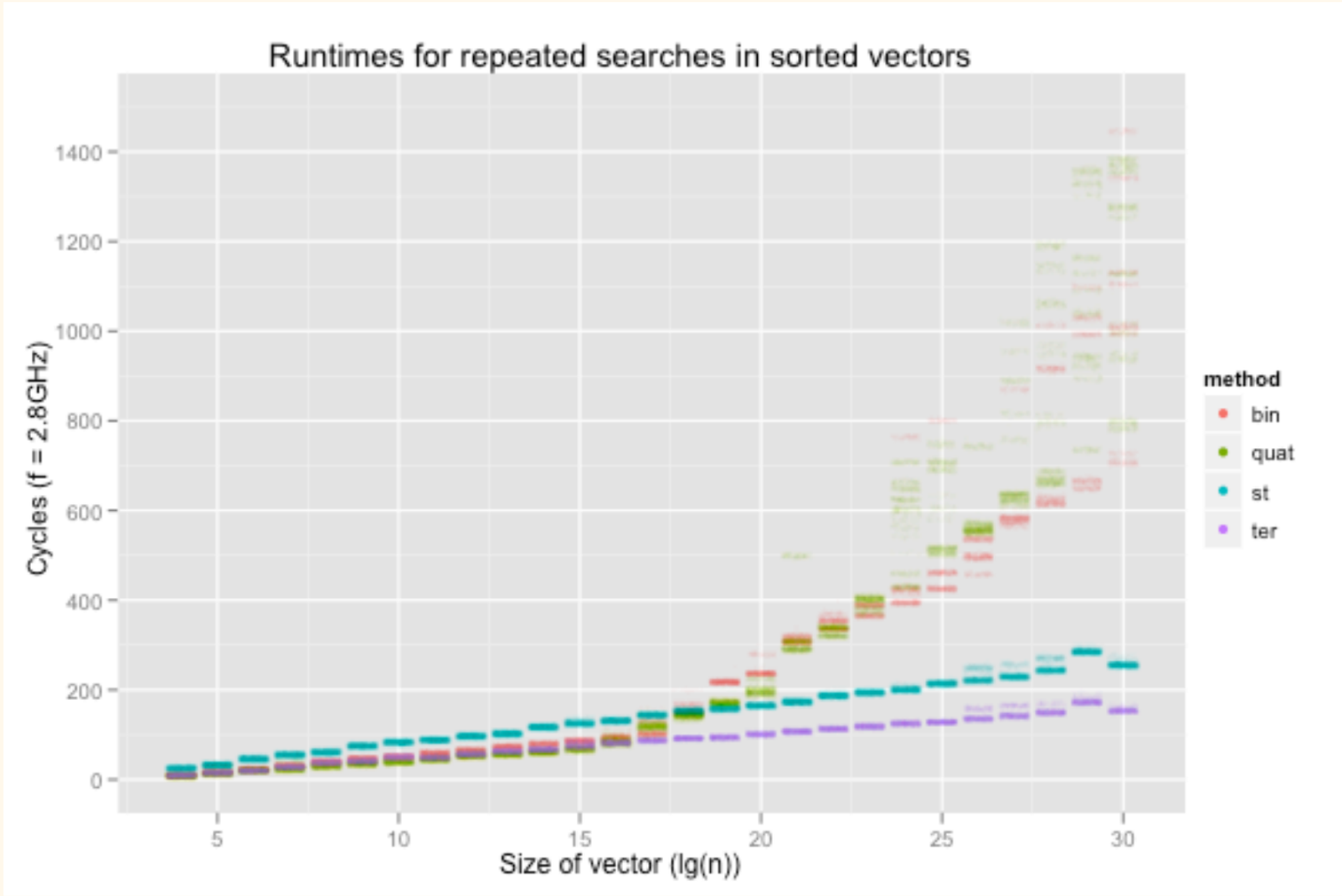
Quick sidebar on caching...

Don’t forget about caching...



http://www.pvk.ca/Blog/2012/07/30/binary-search-is-a-pathological-case-for-caches/

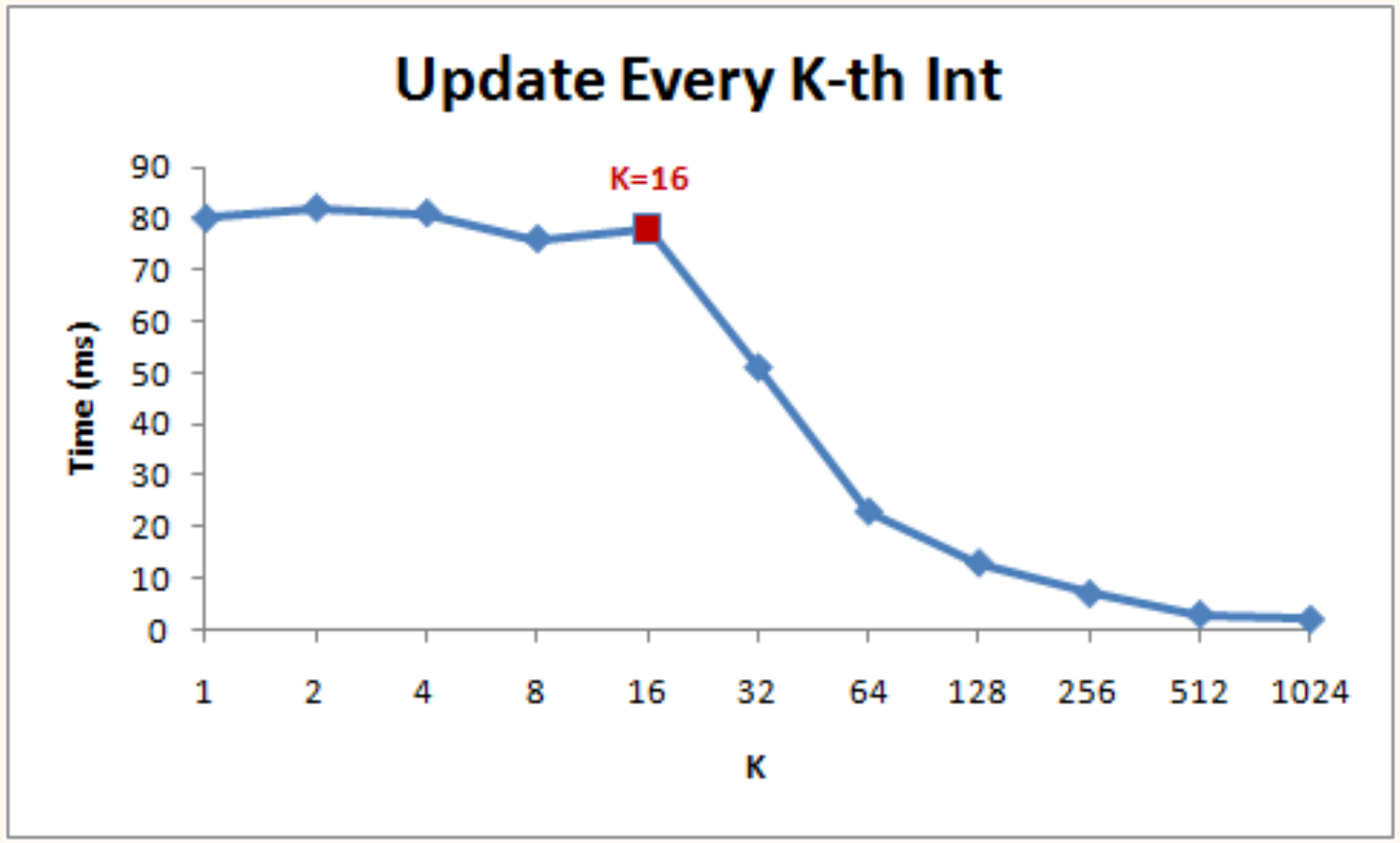
Don’t forget about caching...



http://www.pvk.ca/Blog/2012/07/30/binary-search-is-a-pathological-case-for-caches/

Don’t forget about caching...

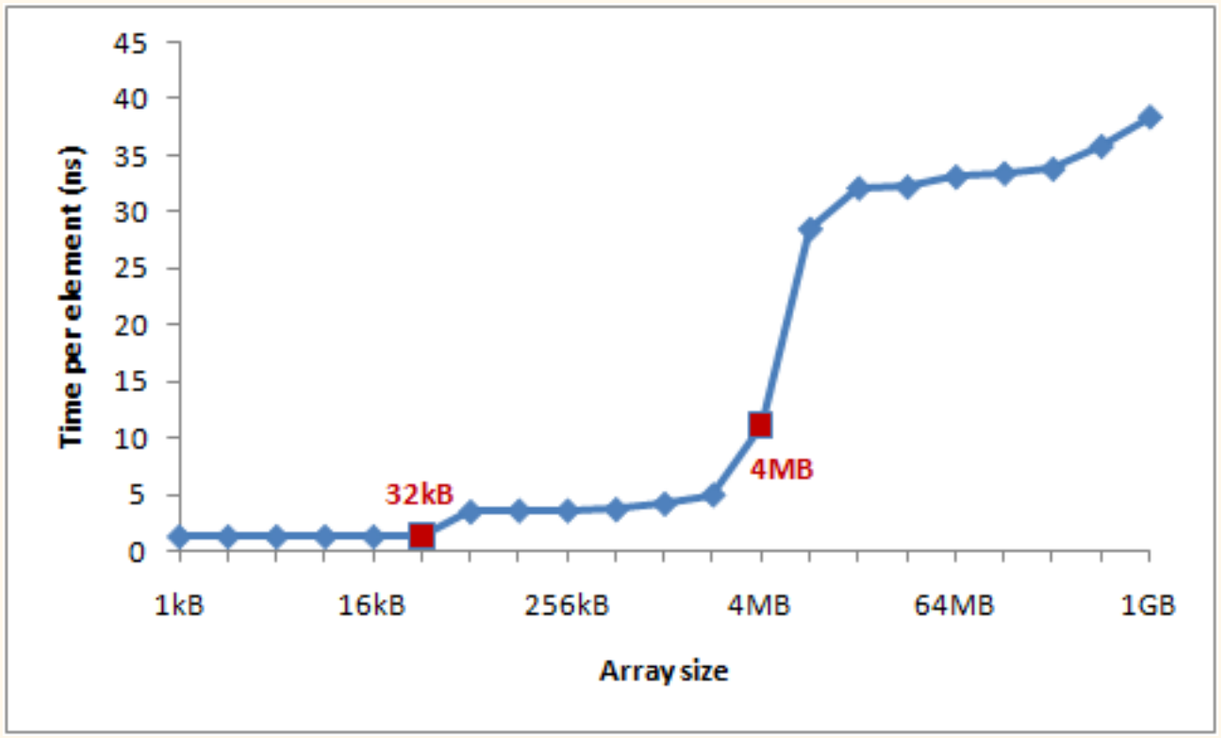
```
for (int i = 0; i < arr.Length; i += K) arr[i] *= 3;
```



http://igoro.com/archive/gallery-of-processor-cache-effects/

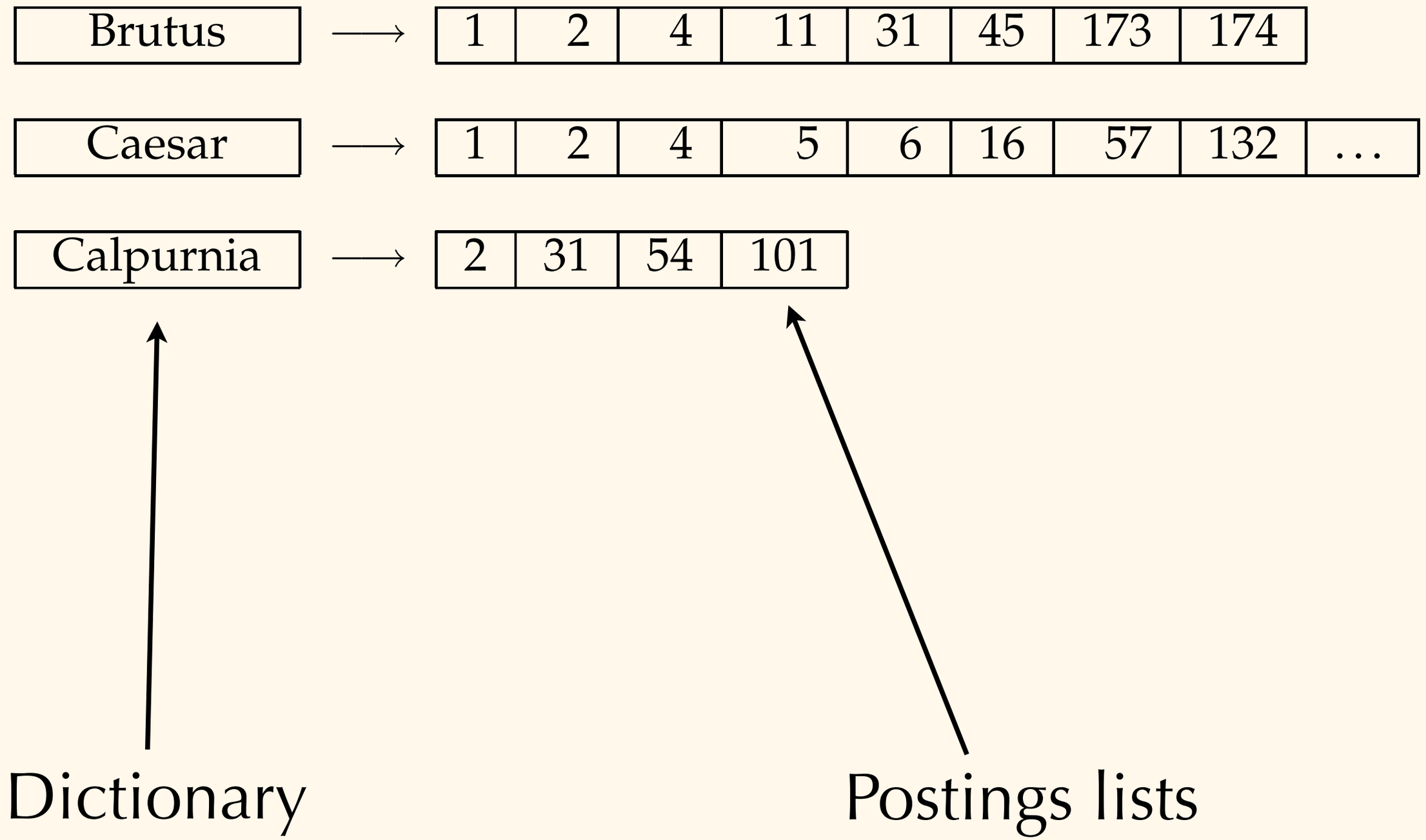
Don’t forget about caching...

```
int steps = 64 * 1024 * 1024; // Arbitrary number of steps
int lengthMod = arr.Length - 1;
for (int i = 0; i < steps; i++)
{
    arr[(i * 16) & lengthMod]++; // (x & lengthMod) is equal to (x % arr.Length)
}
```

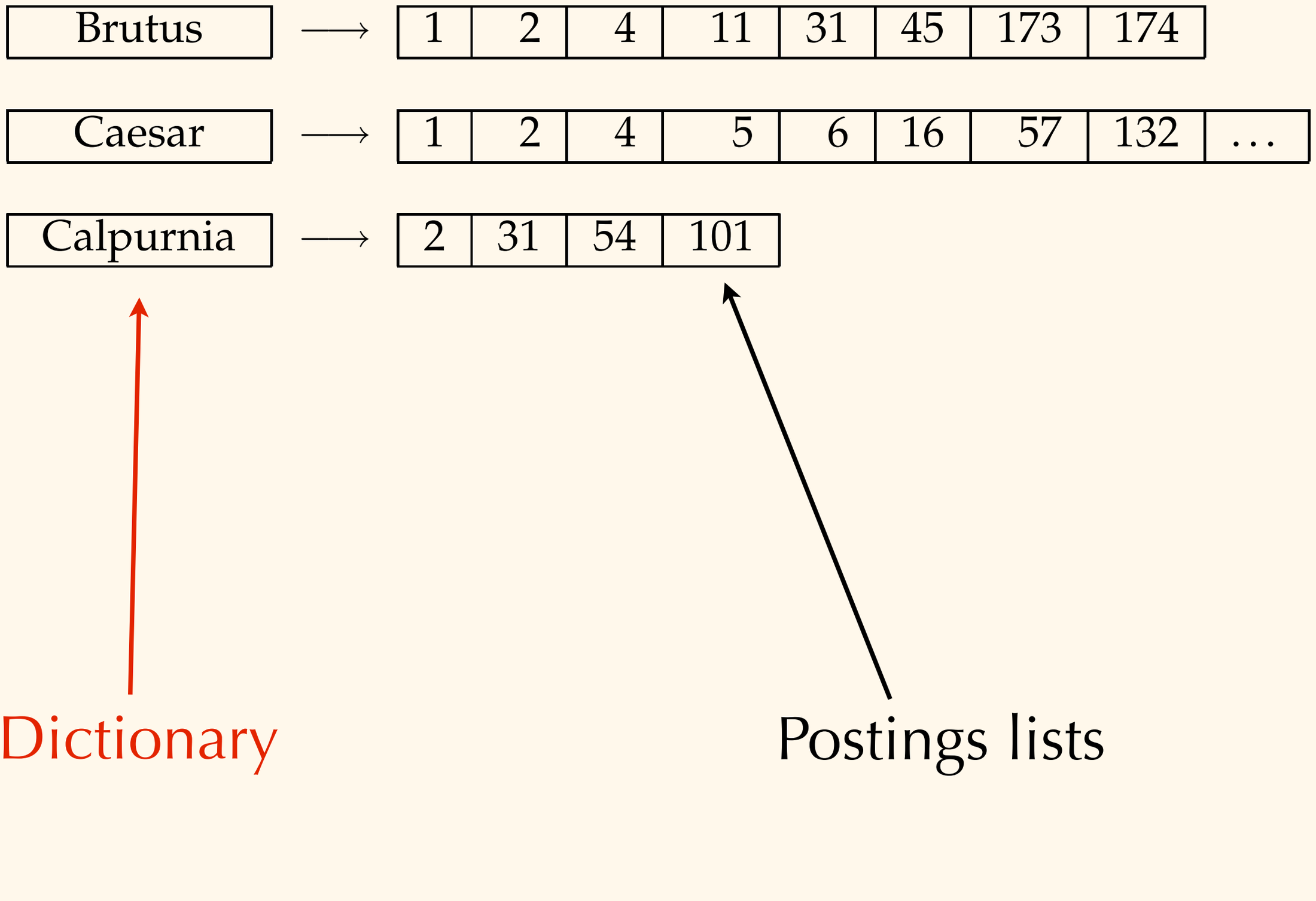


http://igoro.com/archive/gallery-of-processor-cache-effects/

There are two ways to compress an index:



There are two ways to compress an index:



Pre-processing is one approach to dictionary compression:

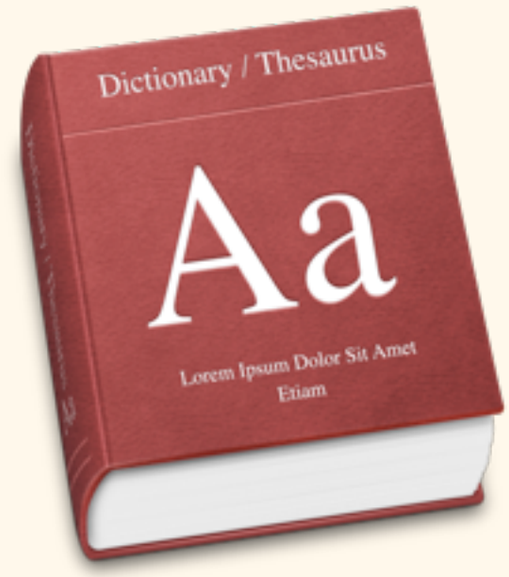
	(distinct) terms		
	number	Δ%	T%
unfiltered	484,494		
no numbers	473,723	−2	−2
case folding	391,523	−17	−19
30 stop words	391,493	−0	−19
150 stop words	391,373	−0	−19
stemming	322,383	−17	−33

Fewer dictionary terms == smaller dictionary, fewer posting lists, etc.

Note that this is language-dependent!

	(distinct) terms			nonpositional postings		
	number	Δ%	T%	number	Δ%	T%
unfiltered	484,494			109,971,179		
no numbers	473,723	−2	−2	100,680,242	−8	−8
case folding	391,523	−17	−19	96,969,056	−3	−12
30 stop words	391,493	−0	−19	83,390,443	−14	−24
150 stop words	391,373	−0	−19	67,001,847	−30	−39
stemming	322,383	−17	−33	63,812,300	−4	−42

How to estimate the number of terms in a collection?



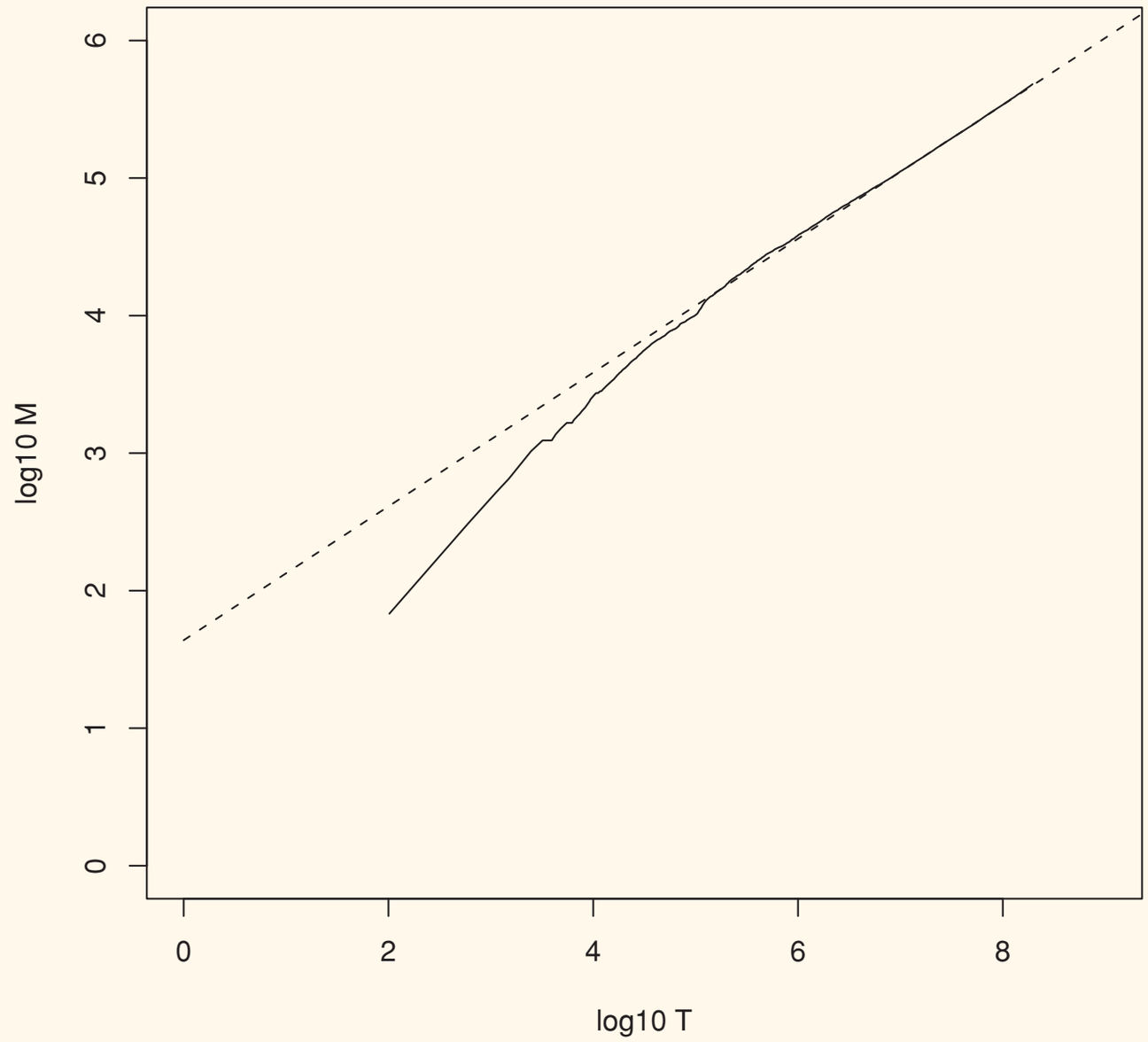
Counting the number of distinct words in, say, the OED is a tempting way to start...

... but often results in dramatically under-estimated counts.

(Think names of places, products, genes/proteins, etc.)

How to estimate the number of terms in a collection?

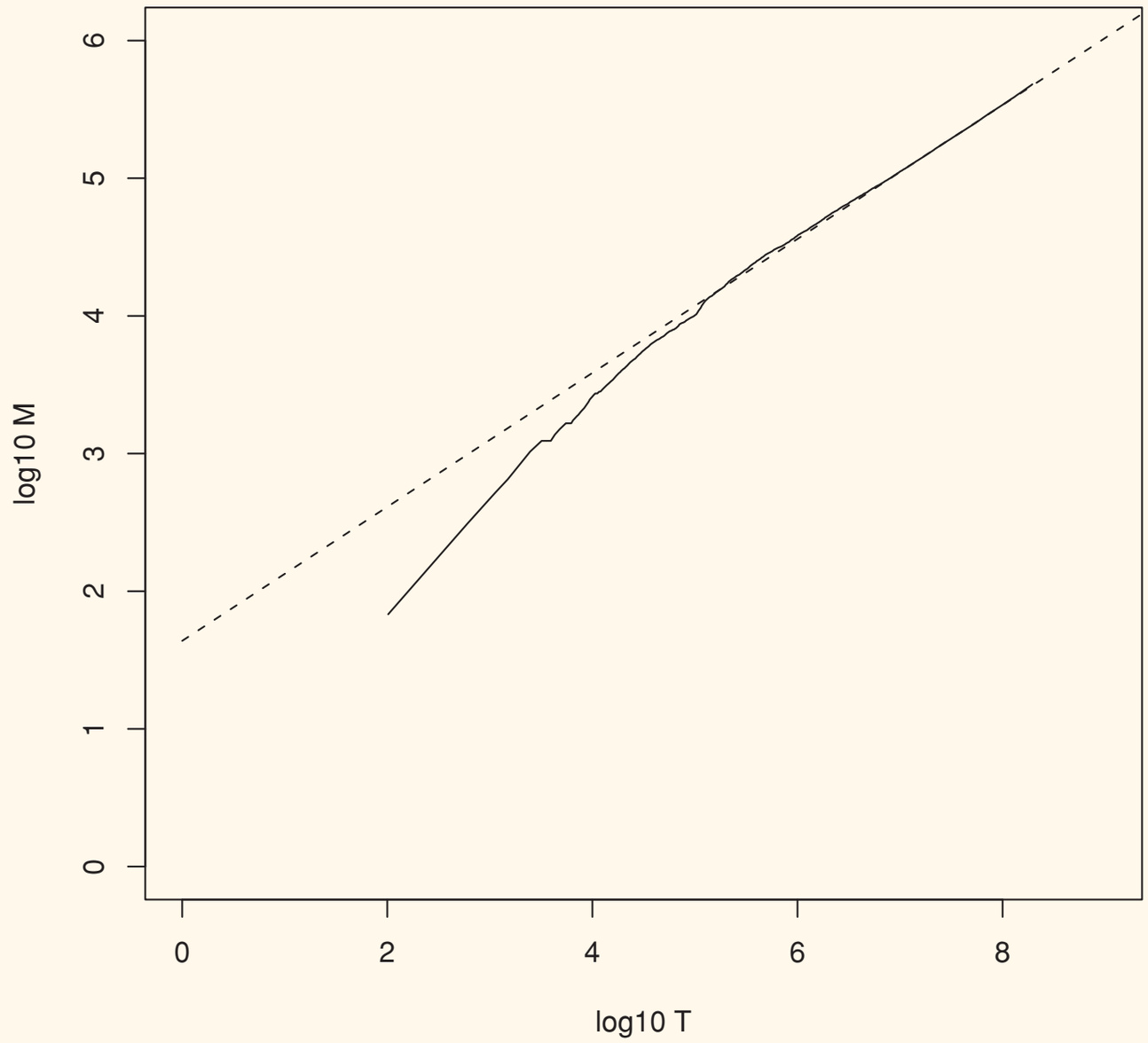
$$M = kT^b$$



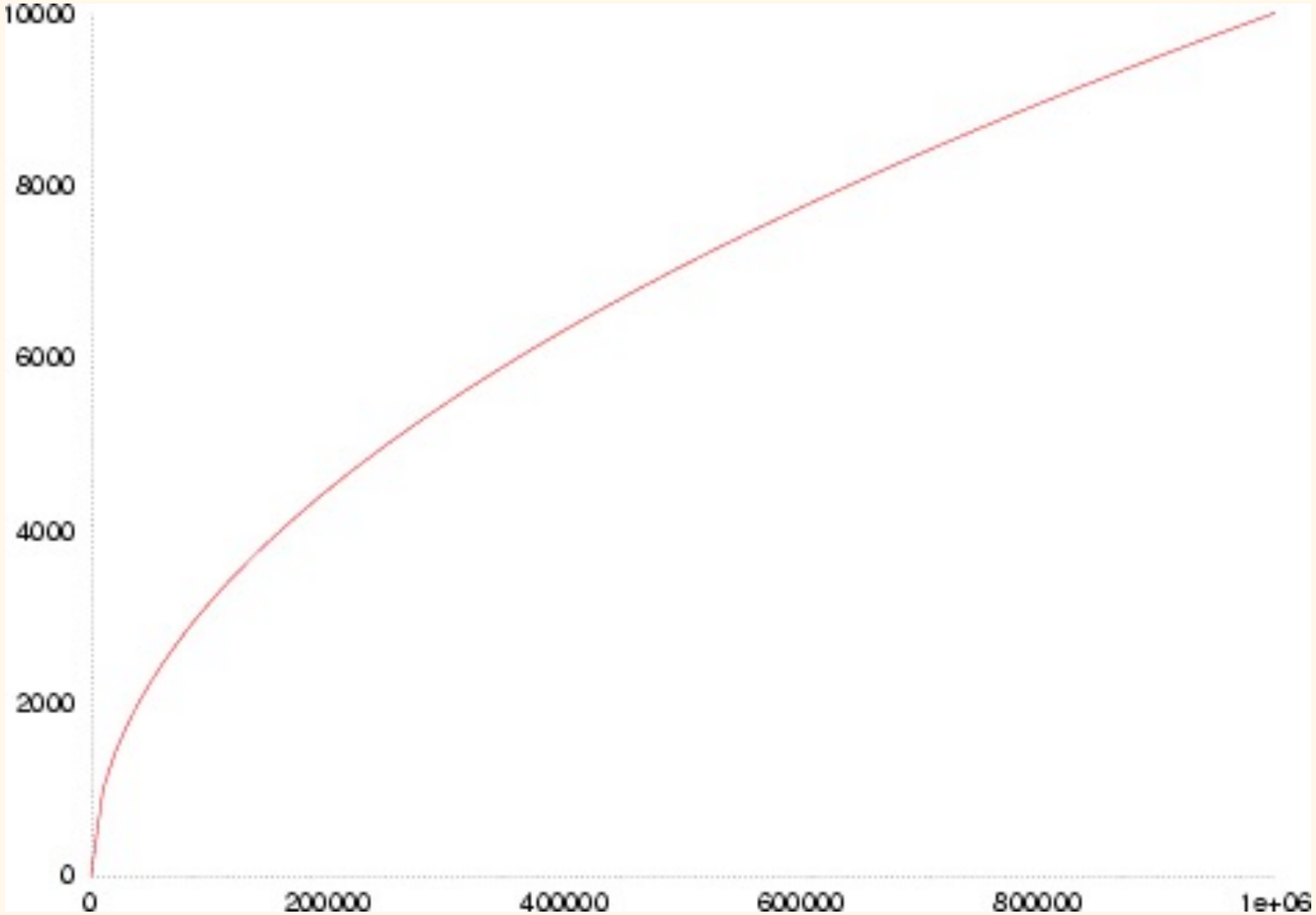
Heaps' law curve for vocab size *M* in collection of size *T* tokens.

How to estimate the number of terms in a collection?

$$M = kT^b$$

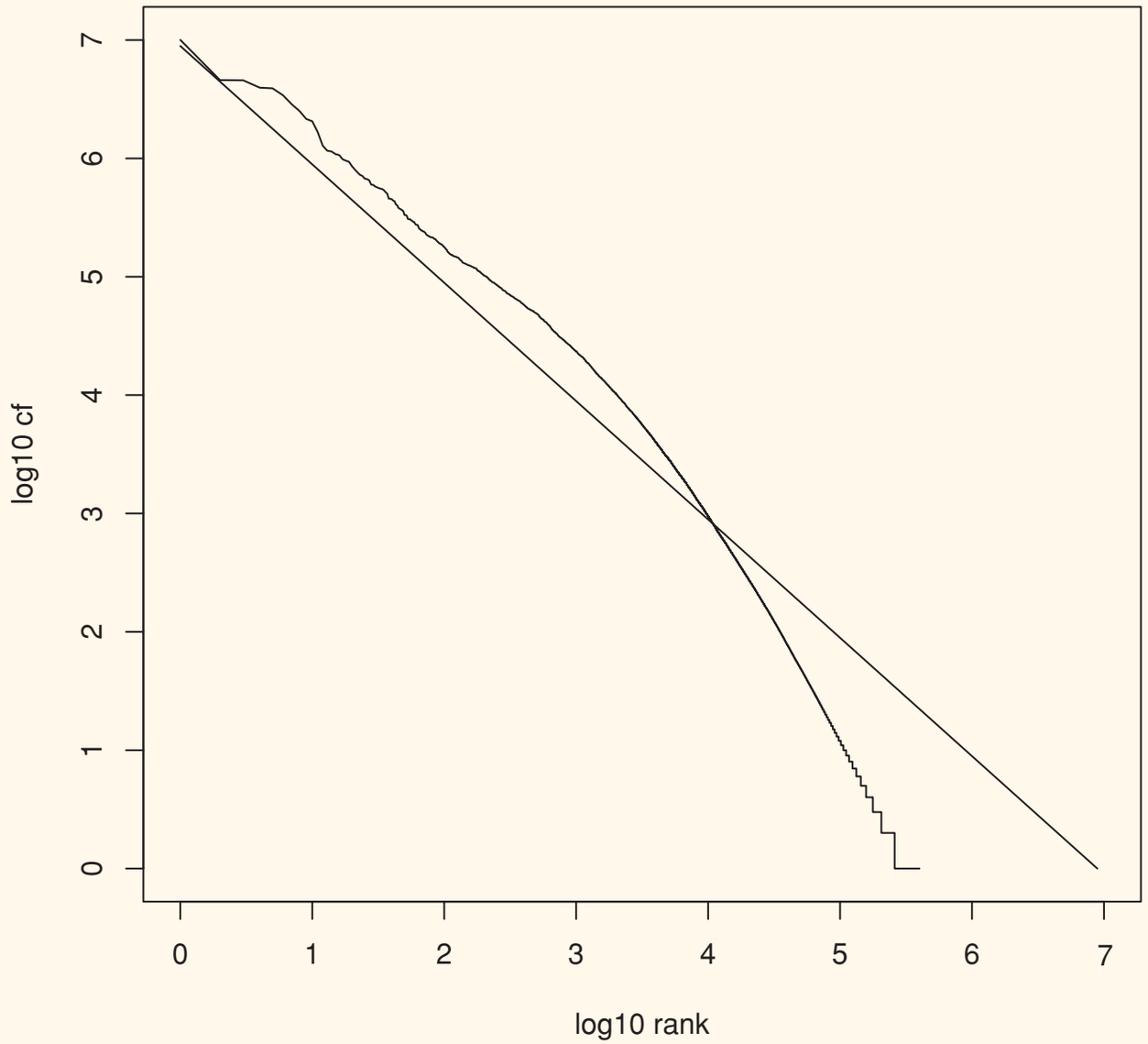


Implication: M increases continually (i.e., doesn't plateau once the collection gets to a certain size).



https://en.wikipedia.org/wiki/Heaps'_law

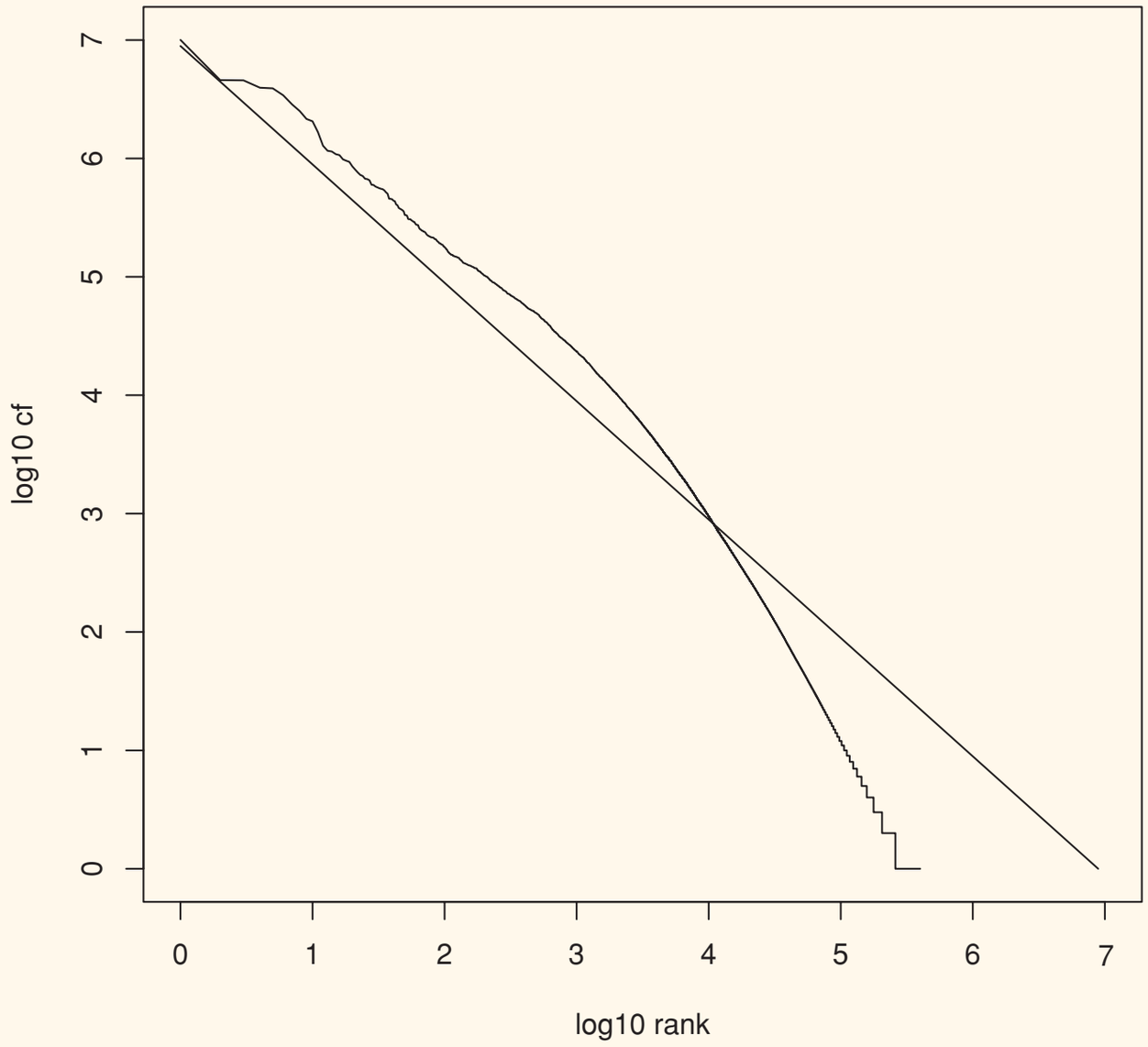
What about term distribution within collection?



$$\mathrm{cf}_i \propto \frac{1}{i}$$

Zipf's law: collection frequency of a term decreases rapidly with rank.

What about term distribution within collection?



Implication: A small number of terms are very common; most are rare.

The point of dictionary compression:

Fit as much of the dictionary as possible in main memory.

Because of Heap's law, large collections will have large dictionaries...

... and many search engines are multilingual!

Warning: here there be pointers...

Warning: here there be caveats...

#1: For the rest of today, we shall pretend that all text is ASCII.

Warning: here there be caveats...

Also: the book uses a 32-bit address space. Large collections need more.

The simplest possible dictionary structure:

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

In RCV1*, 11.2 MB needed to store 400,000 dictionary entries.

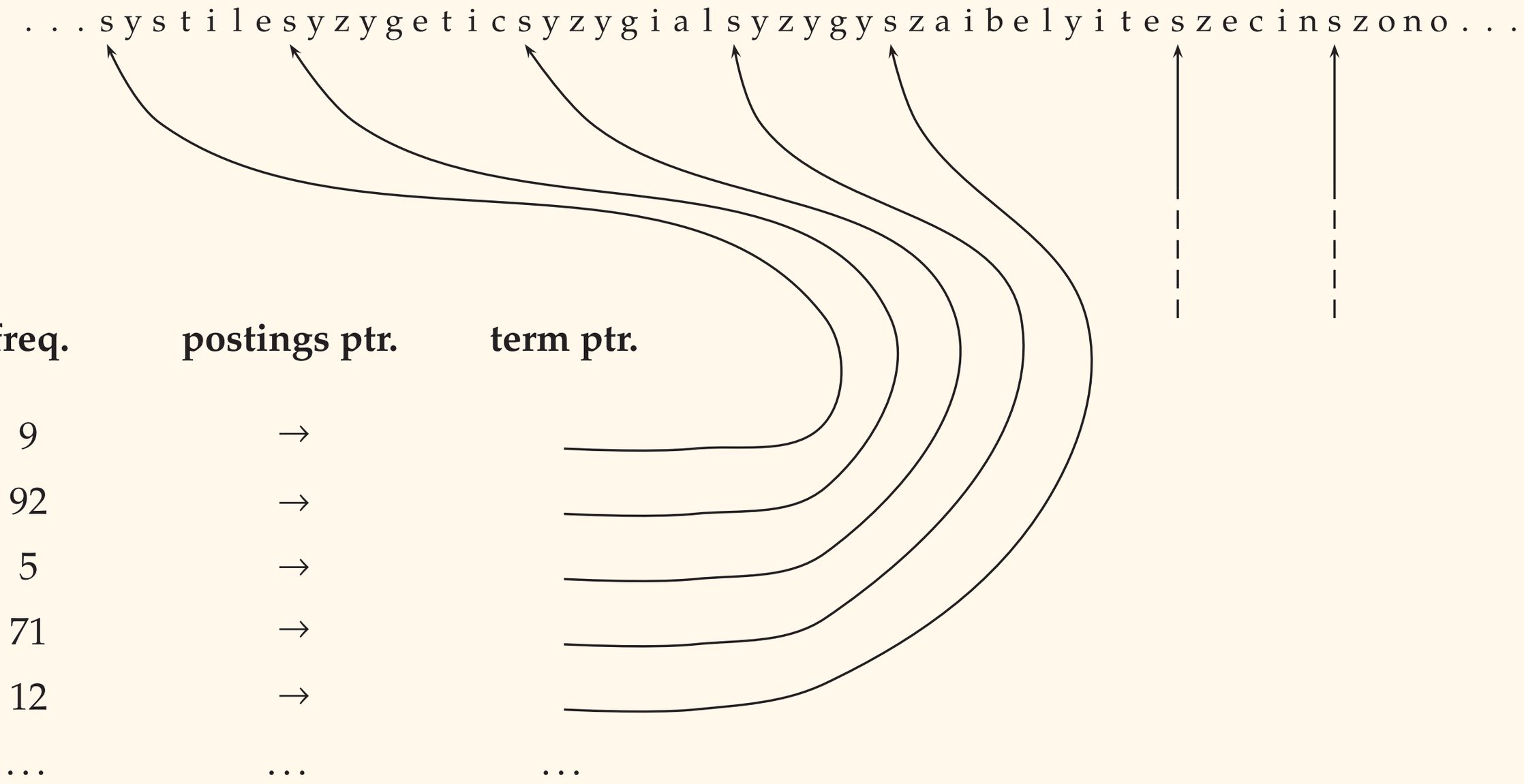
RCV1: "Reuters Corpus Volume 1," a newswire corpus.

The simplest possible dictionary structure:

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

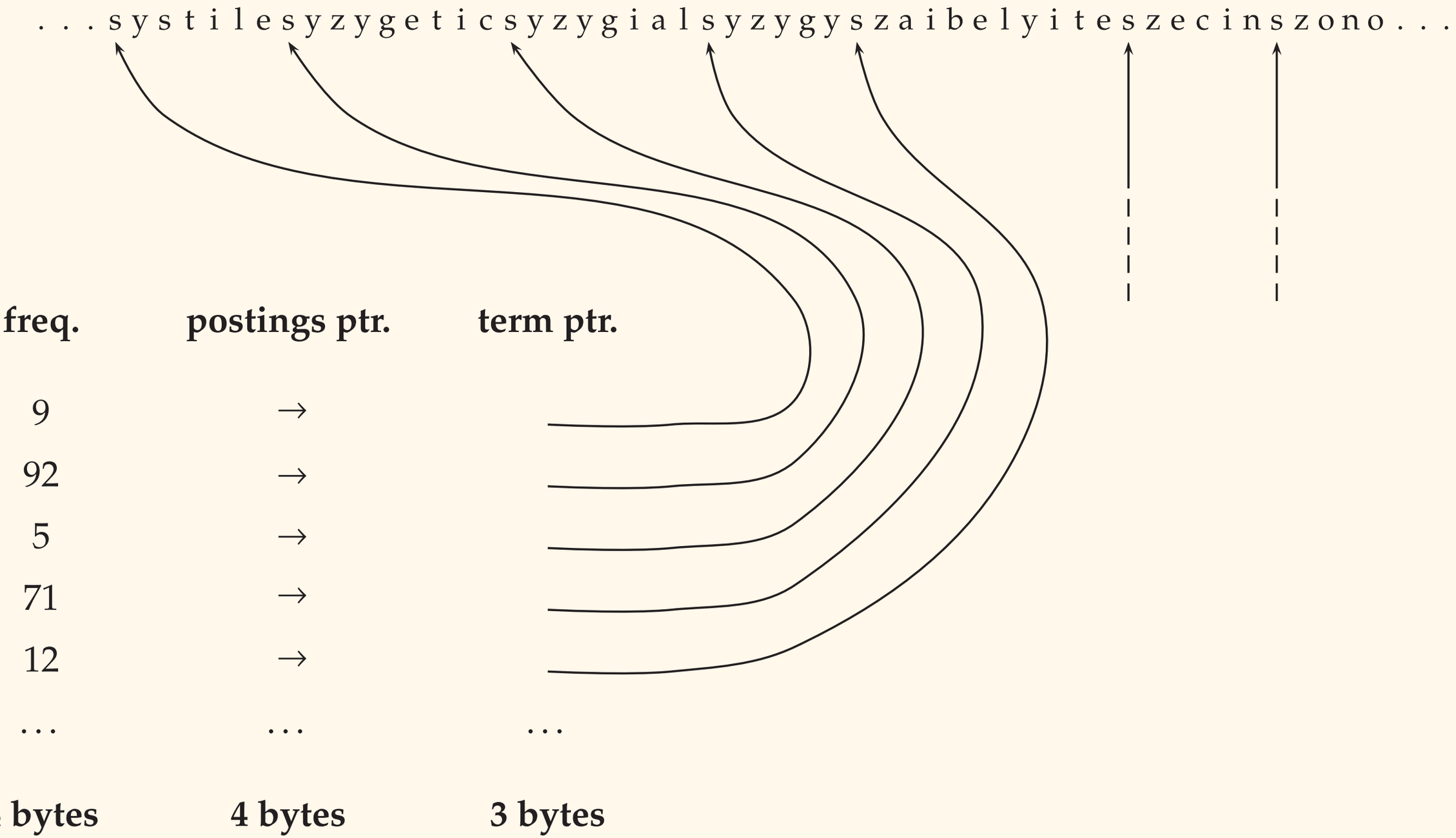
Fixed-width entries are both wasteful and limiting, but are simple to implement.

Next: dictionary-as-a-string



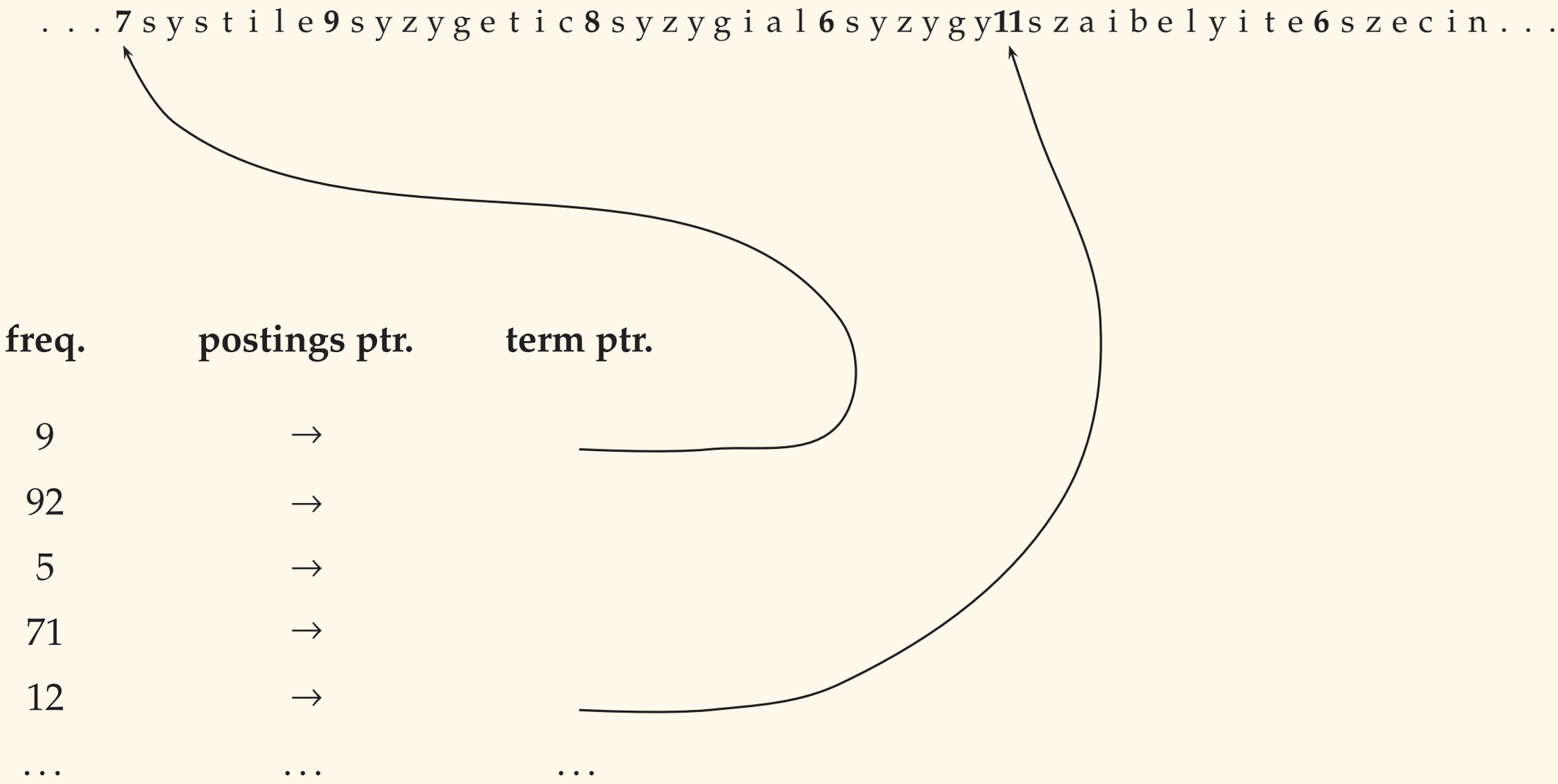
In RCV1, 7.6 MB needed to store 400,000 dictionary entries.

Next: dictionary-as-a-string



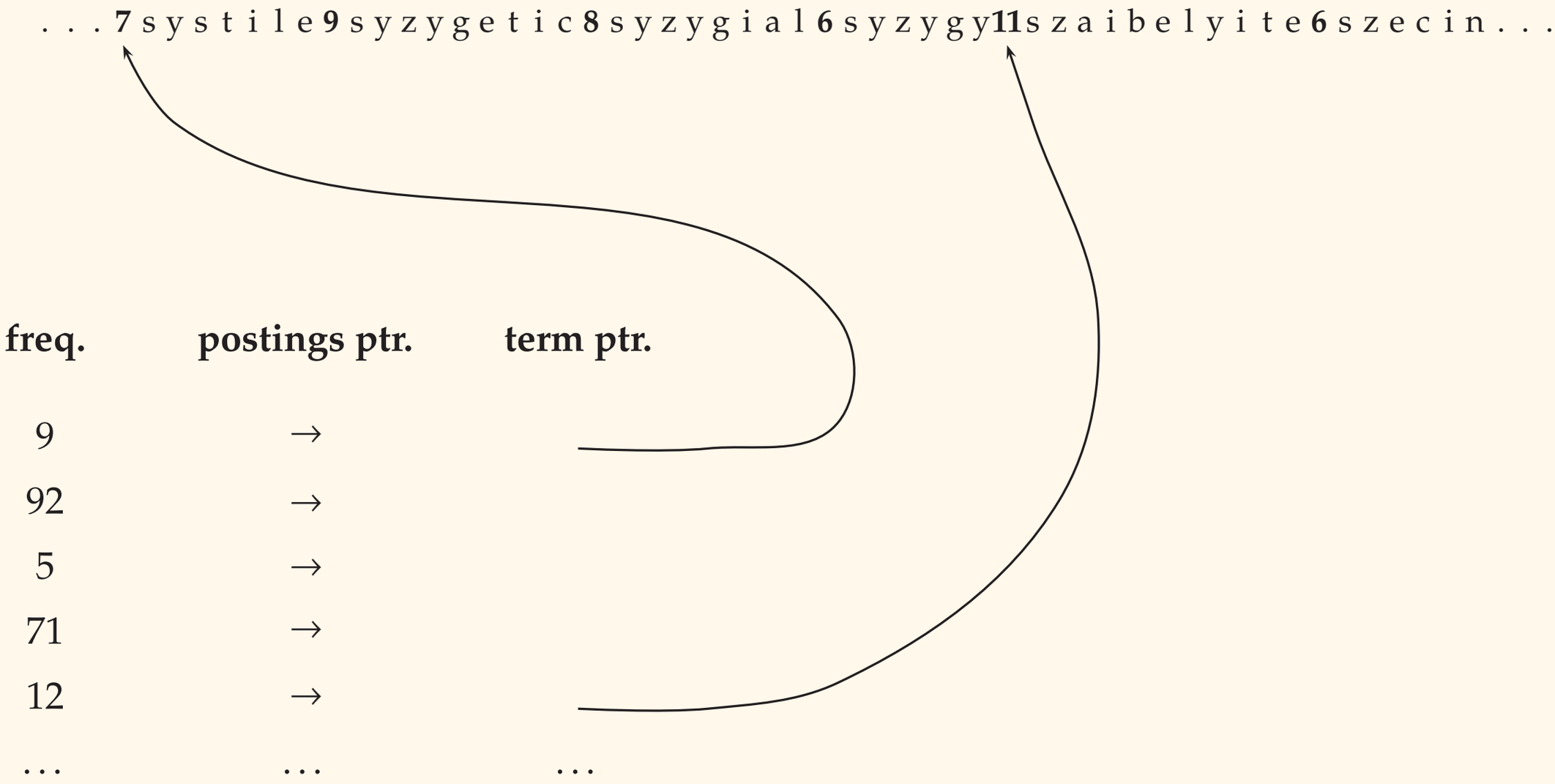
Some of the space saved by the variable width is offset by the need for term pointers.

Blocked storage:



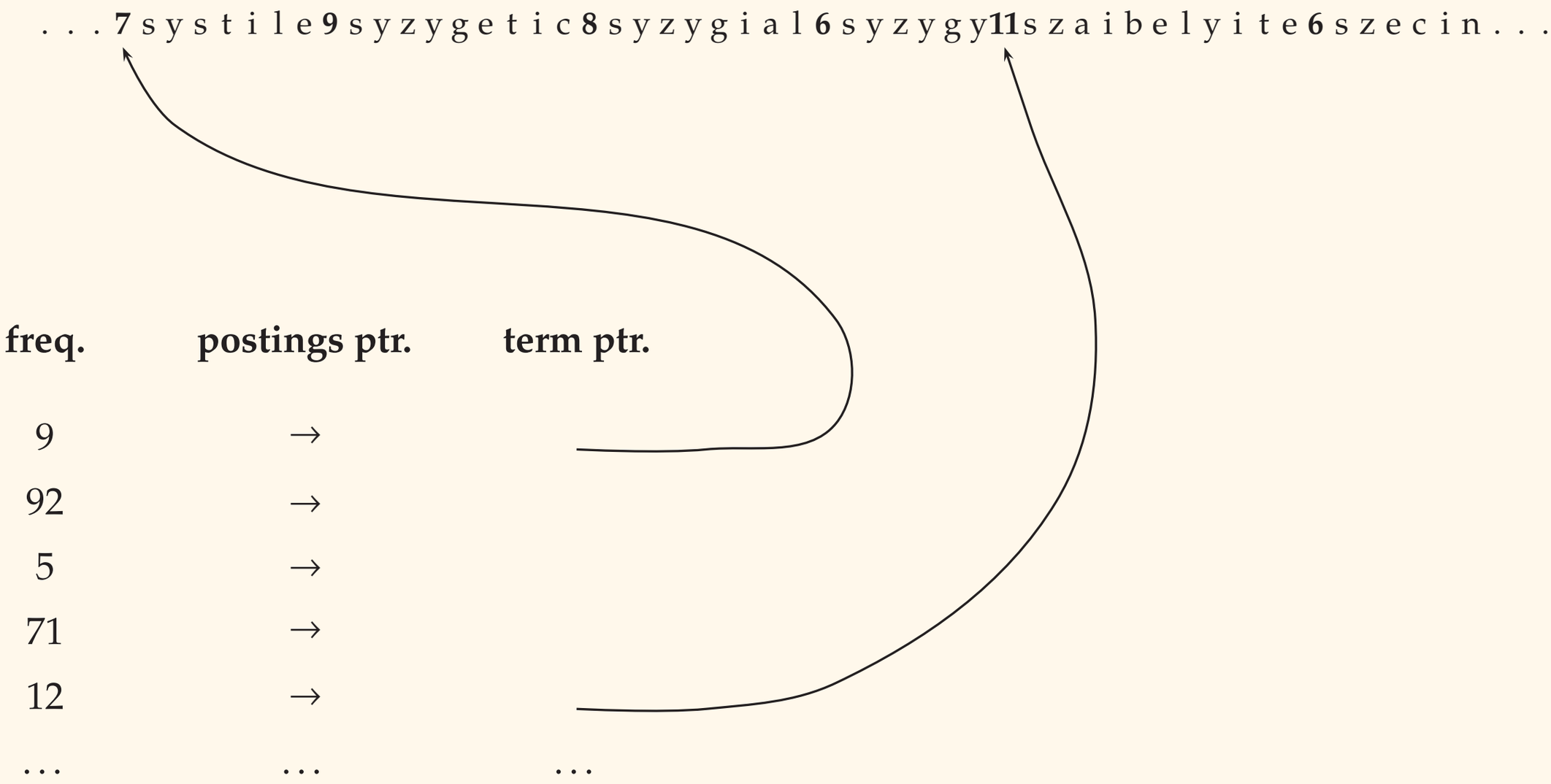
Pick blocks of size k, and only store pointer to first term of each block. Add in-band term lengths to dictionary string.

Blocked storage:



This saves *k - 1* term pointers, but adds *k* bytes for term lengths.

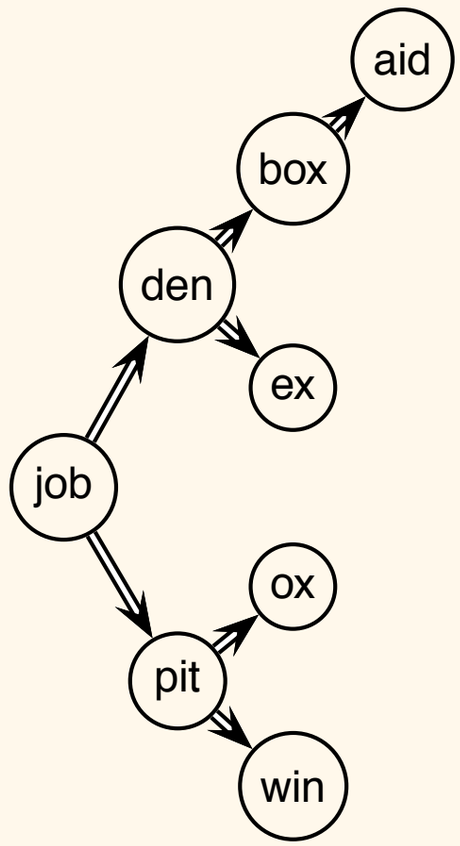
Blocked storage:



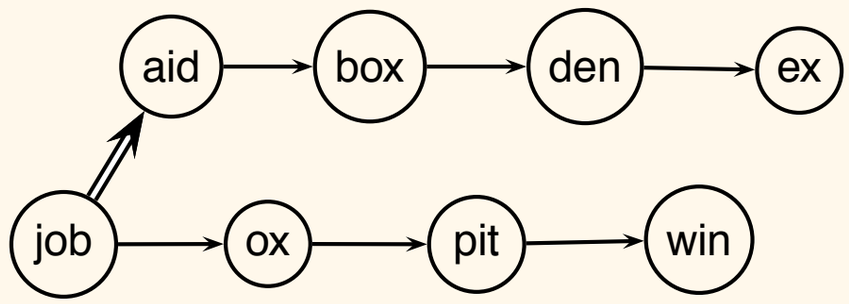
For RCV1, we are now down to 7.1 megabytes.

But there is always a tradeoff: Term lookup now takes more time.

(a)



(b)



Seeking through the uncompressed dictionary involves on average 25% fewer steps.

Front coding takes advantage of common prefixes to save space.

One block in blocked compression ($k = 4$) ...
8automata**8**au t omate**9**au t omatic**10**automation

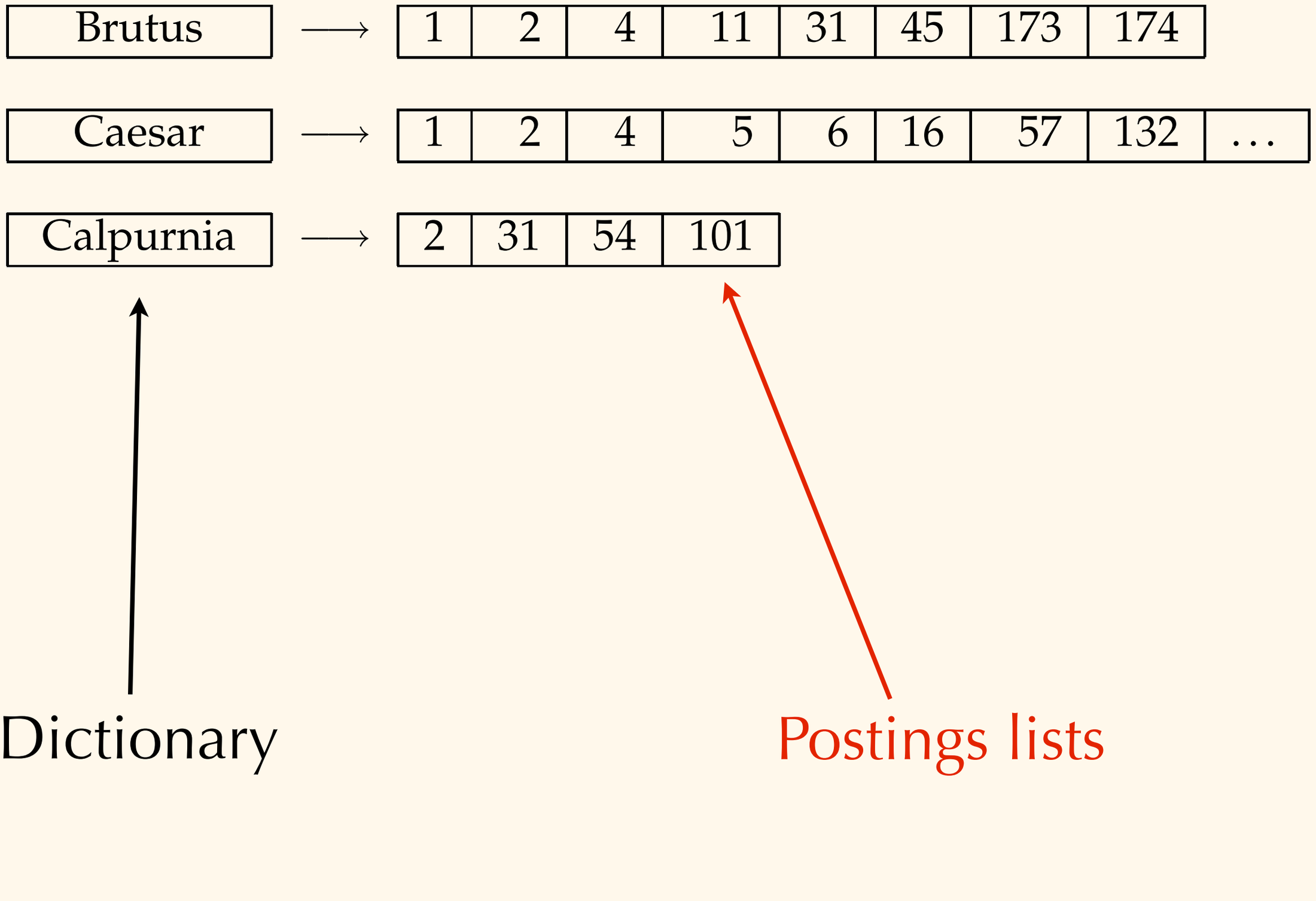
⇓

... further compressed with front coding.
8automat∗a**1**⋄e**2** ⋄ i c**3**⋄i on

Dictionary compression for Reuters-RCV1.

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

There are two ways to compress an index:



Simplest approach to posting list:

Store lists of complete docIDs.

RCV1 has 800,000 documents, so we need $\log_2 800,000 = 20$ bits per docID.

Approximately 250 MB uncompressed.

800,000 is tiny; bigger collections need more bits per docID (many more).

Key observation: postings for frequent terms are often close together in the collection.

What if we store *gaps* or *offsets* between docIDs rather than docIDs themselves?

	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						

Many words wouldn't need a full 20 bits to be represented...

We can use a variable byte code to more efficiently use space:

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Figure 5.8 in the book gives an example algorithm...

Using this scheme achieves >50% reduction in posting list space (down to 116 MB).

In practice, these schemes can be applied to different units than bytes (16-bit words, etc.).

Variable-byte encodings are simple and work well... but can we do better?

Yes, by using bit-level encodings (like the **γ** encoding).

But is it *enough* better to be worth the significant hassle? Probably not.

(Also note that response time depends in part on time spent seeking through index)

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
term 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

Next up: Experimental evaluation.